UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

Barcelona
Supercomputing
Center
Centro Nacional de Supercomputación

Scuola di Ingegneria Industriale e dell'Informazione
Dipartimento di Elettronica, Informazione e Bioingegneria
Corso di Laurea Magistrale in Electronics Engineering

POLITECNICO
MILANO 1863

Assessment of a microcontroller for safety-critical avionics and automotive systems

Advisors:
Dr. Luis J. de la Cruz Llopis - UPC
Dr. Jaume Abella Ferrer - BSC
Prof. William Fornaciari - Polimi

Master thesis of:
Lorenzo Giuseppe Toscano
Matr. 884758

Academic Year 2017-2018

Acknowledgements

I would like to dedicated this page to thank many people that I met during my Master thesis period that made this experience unforgettable.

Firstly, I want to thank Jaume Abella Ferrer, my advisor at BSC, for the helps that he gave me. He followed me during all my the Master thesis period, giving me the materials that were needed to study for the purpose to achieve. Thanks for the patient that he had to answer all my questions and my doubts that arisen during this experience at BSC, for his kindness and for the time that he dedicated to help me.

I'm grateful to Mikel Fernández, engineer at BSC, for his help and for the tips that he gave me about the several new programming languages that I had to deal with. Thanks for his patient, for the time he spent to explain me clearly new concepts about my work, speeding up all the work that I had to do, and to repeat some of them that I didn't understand the first time.

Particularly thanks go to Dr. Enrico Mezzetti, which help me a lot with many tips about programming languages and technical documents, and to the PhD student Suzana Milutinovic, which helped me to understand some new concepts during my work. I'm grateful to Dr. Francisco J. Cazorla, leader of this group, which introduced me to all the CAOS team members the first day that I arrived, and to Dr. Leonidas Kosmidis for his advises and helps that he gave me during my path to conclude this work.

Thanks very much to Matteo Fusi for his help and the contribution that he gave for this Master thesis.

Thanks to the whole CAOS team, which embraces me in their team with great politeness and congeniality, making feel me part of it from the first week that I started to work.

Last thanks are dedicated to my friends and to my family, from my parents to my grandfathers, which believed in me and they supported me each day, cheering me up when problems seemed to be too much difficult to be overcame.

Thank you all,
Lorenzo

Abstract

Nowadays, microcontrollers used in critical real-time embedded systems use mostly one core, but are being replaced with more powerful hardware platforms that implement multicore systems. Among the latter, it is possible to identify in the space domain, for instance, the Cobham Gaisler NGMP developed for the European Space Agency (ESA), which is built with a SPARC quad-core processor that has a two-level cache hierarchy. For what concerns automotive and avionics environments, very flexible platforms like the Zynq UltraScale+ EG one has been regarded as a very powerful platform for these high-performance safety-critical systems. In fact, the aforementioned Zynq board implements two multicore clusters, namely an ARM dual-core Cortex R5 and an ARM quad-core Cortex A53, as well as a GPU and an FPGA. Due to the industrial trend towards the deployment of autonomous driving in the automotive domain and unmanned vehicles in the avionics domain, boards with such multicore systems are very promising.

The use of multicores brings a concern related to contention (interference) in the access to shared hardware resources, which challenges timing verification needed to prove that all critical real-time tasks will execute by their respective deadlines. In particular, Worst-Case Execution Time (WCET) estimates for tasks need to account for the impact in execution time that contention in shared resources may have. While such analysis has been performed on relatively-simple multicores, like the NGMP, it needs to be carried out on the more powerful and complex Zynq UltraScale+ EG platform. In particular, it is required to analyze the different sources of interference for the multicore

clusters and how tasks need to be consolidated so that resource sharing is performed efficiently across tasks, thus minimizing the impact on execution time for the most critical real-time tasks.
In this Master thesis work, the measurement-based methodology developed at Barcelona Supercomputing Center (BSC) to quantify the interference that arises across cores due to contention in shared hardware resources, is ported from the (simple) NGMP platform to each of the computing clusters of the Zynq UltraScale+ EG platform. Such methodology consists in the use of small microbenchmarks that aim at stressing specific shared hardware resources to create very high contention. Hence, this thesis investigates how to produce high contention in the shared hardware resources of the Zynq UltraScale+ EG platform, thus integrating those concepts working on the SPARC V8 instruction set of the NGMP to the ARM v7 and ARM v8 instruction sets of the Zynq platform. This requires porting and adapting microbenchmarks written partly in assembly code, verifying the Performance Monitoring Unit, and analyzing the sources of contention. As final step, guidelines are devised to properly consolidate software to be implemented on the target platform in order to contain as much as possible interference on critical tasks.

Sommario

Oggigiorno, i microcontrollori utilizzati nei sistemi conosciuti come *critical real-time embedded systems* utilizzano principalmente un core, ma tendono sempre di più ad essere sostituiti con piattaforme hardware più potenti che implementano sistemi multicore. Tra questi ultimi, è possibile identificare nel dominio spaziale, per esempio, il NGMP Cobham Gaisler sviluppato per l'European Space Agency (ESA), che è stato sviluppato con un processore quad-core SPARC con una gerarchia di cache a due livelli. Per quanto riguarda l'ambiente automotive e quello avionico, piattaforme molto flessibili come quella denominata Zynq UltraScale + EG sono state considerate come piattaforme molto potenti per questi specifici sistemi *embedded* dal punto di vista della sicurezza ad alte prestazioni. Infatti, la Zynq board menzionata precedentemente implementa due cluster multicore, cioè un ARM dual-core Cortex R5 e un ARM quad-core Cortex A53, oltre a una GPU e un FPGA. A causa della tendenza industriale verso lo sviluppo della guida autonoma nel settore automobilistico e dei veicoli senza conducente nel settore dell'avionica, le piattaforme con tali sistemi multicore sono molto promettenti.
L'uso di multicore pone un problema legato alla contesa, ovvero legato all' interferenza, nell'accesso alle risorse hardware condivise, il quale mette in discussione la verifica dei tempi necessaria per dimostrare che tutte le attività che necessitano di essere calcolate in tempo reale (critical real-time tasks) verranno eseguite rispettando le rispettive scadenze. In particolare, le stime del Worst-Case Execution Time (WCET) per le attività devono tenere conto dell'impatto nei tempi di esecuzione che può avere la contesa nelle risorse

condivise. Mentre tale analisi è stata eseguita su multicores relativamente semplici, come il NGMP, essa deve essere eseguita anche sulla più potente e complessa piattaforma Zynq UltraScale+ EG. In particolare, è necessario analizzare le diverse fonti di interferenza per i cluster multicore e come le attività (tasks) devono essere consolidate in modo che la condivisione delle risorse sia eseguita in modo efficiente tra le attività, riducendo così l'impatto sui tempi di esecuzione per le attività più critiche in tempo reale.
In questo lavoro di tesi di Master, la metodologia basata sulla misurazione sviluppata presso l'azienda Barcelona Supercomputing Center (BSC) per quantificare l'interferenza che si genera tra i core a causa della contesa nelle risorse hardware condivise, viene portata dalla (semplice) piattaforma NGMP a ciascuno dei cluster di calcolo della piattaforma Zynq UltraScale+ EG. Tale metodologia consiste nell'uso di piccoli microbenchmark che mirano a stressare specifiche risorse hardware condivise per creare una controversia molto alta. Quindi, questa tesi indaga su come produrre alta contesa nelle risorse hardware condivise della piattaforma Zynq UltraScale+ EG, integrando così quei concetti che lavorano sul set di istruzioni SPARC V8 dell'NGMP ai set di istruzioni ARM v7 e ARM v8 della piattaforma Zynq. Ciò richiede il *porting* e l'adattamento dei microbenchmark scritti in parte in codice assembly, la verifica dei *Performance Monitoring Counters* e l'analisi delle fonti di conflitto. Come passo finale, sono state ideate delle linee guida per consolidare correttamente il software da implementare sulla piattaforma di destinazione al fine di contenere il più possibile l'interferenza nelle attività critiche.

XII

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Nowadays, the demands of high-performance systems are increasing consistently in automotive and avionics domains since industry is adopting platforms that are able to perform increasingly complex functionalities in real-time. For instance, those functionalities related to autonomous driving in automotive and unmanned vehicles in avionics require capabilities for object detection, trajectory prediction, navigation and routing among others, and those capabilities have strict (real-time) deadlines. In fact, many of those systems can be classified as Safety critical systems, meaning that a failure in those systems can cause casualties (or severe injuries), harm the environment or compromise the integrity of the system itself [1]. Failures can be of many types, being them classified mostly as functional and timing. Functional failures correspond to the cases when the system does not perform its expected activities or leads to wrong results (e.g. not braking to avoid running over pedestrians). Timing failures correspond to the cases when the system performs its expected activities too late (e.g. braking too late to avoid the collision against pedestrians). The latter are the focus of this Thesis.
In this chapter, an overall introduction of the work done for this Master thesis is given, defining in particular objectives, procedures and work plan followed. Afterwards, it's explained the Master thesis' structure, highlighting the issues and the arguments addressed in each chapter.

1.1 Objectives

In such Critical Real-Time Embedded Systems (CRTES), it's critical and mandatory to ensure safety in real-time. Due to the need of increasingly high performance, multi-core processors have been adopted to perform all those critical activities within expected deadlines since they provide sufficient levels of performance. However, several challenges arise on the timing behavior due to the effect of inter-task interference in such systems. In fact, when two or more cores are accessing the same hardware shared resources, contention is experienced, reducing the overall performance of the system.
Contention has a direct effect on the execution time of tasks, which may increase. Hence, real-time systems must undergo a validation step to assess to what extent execution time may grow, so that the platform (including hardware and software) can be guaranteed to perform all its activities correctly and timely.
So far, such assessment has been performed mostly on relatively simple multicores such as the Infineon AURIX TC27x architecture for the automotive domain or the Cobham Gaisler LEON4 processor for the space domain. However, autonomous navigation in avionics and automotive requires the adoption of further complex platforms with larger core counts and diverse computation resources (e.g. time-predictable cores, high-performance cores, accelerators).
In this context, the goal of this Master thesis is to assess whether the Zynq UltraScale+ EG platform, a high-performance platform of the interest for several CRTES domains, fits the needs to execute safety-critical real-time software. In particular, this thesis aims at revealing how much execution time may grow due to the contention on the access to shared hardware resources to understand whether it is a suitable platform (and for what type of applications), and how software must be consolidated to make an effective use of the platform.
In order to obtain such information, the aim is generating as much pressure

as possible on specific resources of the Zynq UltraScale+ EG platform with stressful workloads provided by specific codes that are devised for such purpose. Such workloads are intended to expose how much execution time grows when accessing different shared hardware resources with different types of operations. In order to maximize the stress on these specific resources, different types of operations and particular parameters are needed to be employed and the details will be addressed in chapter 5.
In particular, the target of this thesis has been assessing how much such inter-task interference (contention) can affect the performance of the Zynq UltraScale+ EG platform, focusing on the memories cache of both Cortex A53 and Cortex R5 processors, which are implemented in the aforementioned multi-core system. Therefore, execution time has been studied and observed under specific experiments, which aim to stress the cache levels and memory hierarchy implemented in the target processor emulating potential contention scenarios that may arise when consolidating tasks onto this processor.

1.1.1 Requirements

In order to achieve the objectives explained before, some requirements have to be fulfilled by the codes that are devised for the Zynq UltraScale+ EG platform:

- Algorithms have to be written in C/C++ and assembly programming languages. This facilitates binary generation and guaranteeing that binaries perform exactly their intended activities.

- Simplicity and flexibility, meaning that they can easily be implemented in other processor architectures with minimal modifications and, as done in this thesis, can be easily used in different cores with different cache hierarchies with negligible additional effort.

- The amount of memory that their code occupies in the levels caches of the target architecture has to be minimal, so that contention can be

studied with the data patterns accessed without side effects caused by the code footprint.

1.2 Activities

In this Master thesis, the following activities were carried out in order to perform the desired work on the target platform:

- Definition of the work to be done, which is divided into the following steps:
  - Understand clustered architectures and I/O resources implemented on the Zynq UltraScale+ EG platform.
  - Understand the timing behavior of such architecture, focusing on the cluster-related resources and/or I/O interfaces.
  - Build on existing debugger for Zynq to run experiments and collect measurements from such platform.
  - Devise a set of micro-benchmarks[1] that provides empirical evidence of the worst-case and average contention effects.
- Study a priori of the materials given by Barcelona Supercomputing Center (BSC). In particular:
  - Background on the LEON architecture, the AMBA bus interface and the GRMON interface, including the access to the relevant debug/system software.
  - Background on the Zynq UltraScale+ EG architecture.
  - Microbenchmarks. Research papers describing what they do and how they do it.

[1]The starting point was a set of the already existing micro-benchmarks for another architecture, thus using different assembler instructions and different processor parameters.

1.3 Work plan

The development of this work was organized in order to conclude it in four months approximately, following the schedule represented below:

- Reading documentation and getting familiar with the hardware platform: March
- Porting and development of microbenchmarks: April - May
- Evaluation and Master thesis writing: June

By the way, further time was needed to perform and conclude correctly all the steps described so far. Therefore, this led both to perform experiments on the target platform and to write this Master thesis also in July.
Details on the schedule described so far are addressed in the section below, underlining the names of each task and the time spent to complete them.

1.3.1 Gantt chart

The tasks followed and performed for this Master thesis are shown in the Gantt chart represented in figure (1.1), while in table (1.1) are represented the names of each of them.
In the last week of February, it was defined the work plan of this Master thesis described previously in section (1.3) (*Work plan definition*) and materials were provided by BSC in that period. Such documentation was studied from the beginning of March until the end of the same month. As shown in the chart (1.1), the largest part of the time was spent to devise the microbenchmarks to be performed on the single cores, which was approximately between the beginning of April and the middle of May. Then, experiments on single cores were ran between middle of May until almost the end of June, while the final ones were performed on multi-cores until middle of July.

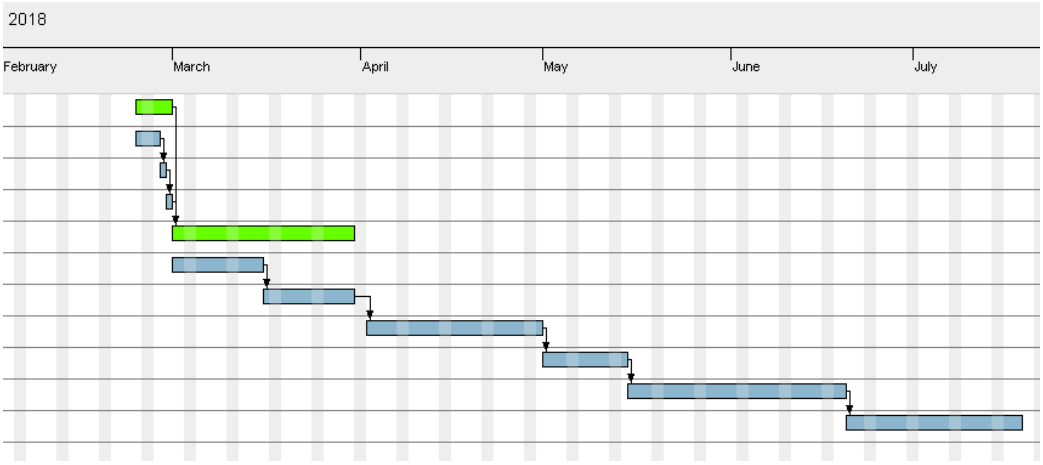

Figure 1.1: Gantt chart

#	Tasks names
1	Work plan definition
2	Arguments definition
3	Work time definition
4	Materials provided by BSC
5	Study before implementations
6	Microbenchmark study
7	SPARC instruction set study
8	Porting from SPARC to ARM processor architecture
9	Microbenchmark implementation
10	Tests performed on single cores
11	Experiments performed with all the cores in parallel

Table 1.1: Gantt chart legend

1.4 Master thesis structure

Next, the structure of this Master thesis is described.

In the second chapter, some background on key aspects of this thesis are provided. This includes details on how timing validation is performed for CRTES, some information on cache memories structure and their way of working, as well as an overview of the main features of the processor architecture employed by the Zynq UltraScale+ EG platform.

For what concerns the third chapter, it's addressed the state-of-the-art on micro-benchmarks, with particular emphasis on those devised to stress specific timing behavior relevant for timing analysis.

The fourth chapter focuses on describing the methodologies chosen to perform the experiments on the target platform. In particular, it's explained the general working principle for which the micro-benchmarks are devised, and the general schematic of the ones employed in this work are described in details. Moreover, it's described how Performance Monitoring Counters (PMCs) work in the processors like ARM architecture and which ones, relevant for this thesis, are defined in the architectures of the target platform.

In the fifth chapter, the main part of the codes developed and used for running all the experiments on single cores are described in details, starting from the main function of the whole code until the algorithm employed to exploit the PMCs implemented in the aforementioned ARM architectures.

The sixth chapter collects the results obtained with all the experiments performed, considering both the ones obtained on single cores and the ones obtained when all the cores are executing microbenchmarks at the same time. Results are analyzed conveniently reaching relevant research

conclusions.

Finally, in seventh and eighth chapters cost assessment and conclusions are given respectively.

Chapter 2

Background

2.1 Timing analysis

The estimation of the WCET of real-time programs has been investigated for decades. Two main paradigms can be found in the literature on how to estimate the WCET: static timing analysis (STA) and measurement-based timing analysis (MBTA). STA relies on building a timing model of the processor on which to perform abstract interpretation of a structural representation of the program to be analyzed to predict how much each instruction (and hence the whole program) can take to execute, without actually executing the program.

In particular, the timing model of the target processor architecture is built identifying each hardware component, as well as their behavior and relationships in what refers to timing behavior. For instance, cache memories are modelled, including their contents and so, whether a given access would hit or miss. Then, the representation of the source code of the program (in the form of assembly instructions) is analyzed to model both, the execution path flow and the data flow of the program, so that STA can account for the behavior of the different execution paths and potential data values.

In general, abstract interpretation builds upon unknown information such as, for instance, unknown input data values, which may affect memory access patterns and execution paths. Hence, this leads to an explosion of potential states that can be reached after the execution of every instruction. STA makes the problem tractable by making "safe" (i.e. pessimistic) assumptions that allow merging different states into few ones that lead to the highest execution times possible. For instance, if the address accessed by a given load instruction cannot be determined, instead of modelling all potential states corresponding to all potential addresses that could be fetched, STA typically assumes that the access is a miss, that no useful data is fetched into cache, and that some cache contents are evicted (either a cache line or a full cache way). Overall, STA trades complexity and pessimism to keep computational cost tractable. A survey on timing analysis, with particular emphasis on STA, can be found in [2].

However, STA has increasing difficulties with increasingly complex hardware, as analyzed in [3]. In particular, simplifying the analysis process by merging states leads to potentially high pessimism. In general, the higher the hardware complexity (e.g. by using cache memories and multicores), the larger the number of potential states and execution time variation across states, and hence the higher the pessimism to merge states. Moreover, processor timing models are typically derived from processor specifications, which may have thousands of pages, which jeopardizes the reliability of the timing models. Moreover, those specifications are often subject to errata, thus increasing the uncertainty on the reliability of STA [3].

On the other hand, MBTA builds upon execution time measurements of the program under analysis on the target hardware platform to estimate the WCET. MBTA also brings several sources of uncertainty due to the difficulties to guarantee that execution time conditions considered include the WCET or execution time values sufficiently close to it. For instance, generating inputs that trigger the highest number of iterations of loops, the worst paths in conditional constructs (e.g. if-then-else, switch), the worst

memory patterns, etc. is in general out of reach for end users. However, the fact that measurement collection is affordable and WCET estimates built upon measurements are not necessarily overpessimistic makes end users often rely on MBTA [4]. It is common reusing those inputs used for functional test of the software, which typically trigger the different operation modes of software, to obtain execution times relevant for WCET estimation. Then, either by adding an engineering factor to the maximum observed execution time (MOET) (e.g. MOET+20%), or by applying more sophisticated logic (e.g. using some static information about path analysis as done by tools like RapiTime [5]), a WCET estimate is obtained.

However, while MBTA has been proven to be very efficient for single-core processors, multicore processors bring new difficulties due to the potential contention that the task under analysis can experience in the access to shared hardware resources. Thus, specific microbenchmarks causing high levels of contention have been considered to obtain execution time measurements relevant for WCET estimation in multicores [6]. This is the focus of this thesis for a hardware platform – the Zynq UltraScale+ – that brings increasing difficulties due to the use of multiple and heterogeneous core cluster. However, such platform offers high computation power, which is of high interest for many industries such as those in the avionics and railway domains among others.

2.2 Cache memory

One of the more important parameters to evaluate the performance of a multicore processor is the speed access to memory. Low latencies are achieved thanks to the use of cache memories, which are fast enough to serve data and code at high speed. However, such speed is achieved at the expense of making them small enough. Moreover, the introduction of cache memory allows to reduce power consumption and the number of external memory accesses performed by the system to the main memory, which cause slow

downs in the overall system [7].
Typically, the levels of cache implemented in processors aimed to achieve high performances like ARM Cortex-A53 and Cortex-R5 are arranged as represented in figure (2.1).

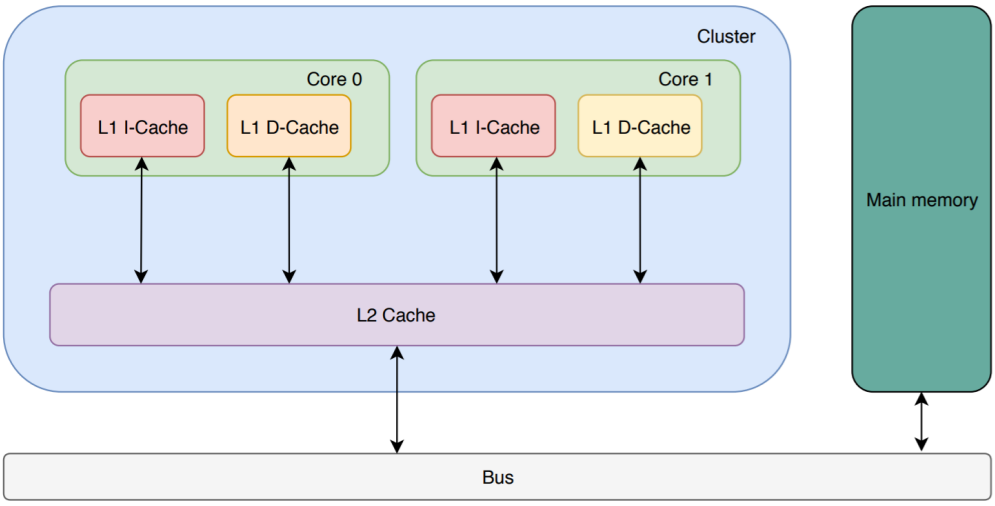

Figure 2.1: General cache memory arrangement

In such figure, two cores are represented as example with two levels of cache, which are the first level (L1) and the second one (L2). Note that L1 cache is divided into Instruction Cache (I-Cache) and Data Cache (D-Cache), namely a modified Harvard architecture within which instruction and data buses are separated in order to reduce interference among them building on the fact that instruction and data access streams are naturally decoupled in program execution. The level-2 cache is a resource typically shared among several cores (e.g. within a cluster of cores) and it receives and sends both instruction and data to the different cores.
When a data is required for the first time, there is no improvement in terms of access time to the memory since that it is not yet present in the cache system. Therefore, the first time some data is accessed, the data will be fetched from the Main Memory block, and it will go through the interconnection network (e.g. an AMBA AHB processor bus) to reach the most internal level cache,

typically being stored first in the shared L2 cache and then in the L1 Cache. Subsequent accesses to the same data will be much faster since it is already stored in the internal cache and there is no need to fetch such data from the main memory [7].
Depending on the fact that data is found or not in the cache, the two previous cases can be distinguished as follows:

- **Cache hit** is the case when the data is found in the cache, allowing fast accesses.
- **Cache miss** corresponds to the case when the data is not found in the cache and it has to be sought in higher cache levels or directly in the main memory, namely the highest memory level. Then, such data has to be copied in the cache. These steps lead to slow memory accesses, reducing system performance.

2.2.1 Cache structure

The types of cache structures that can be found in processors like the ones implemented in the Zynq UltraScale+ EG platform considered in this Master thesis are the following ones:

- Cache Fully Associative, thanks to which each location in main memory can be stored in any position of the cache. In general, allowing any data to be placed in any cache location requires expensive searches upon an access to determine whether the data is available in cache or not. Hence, this type of caches is expensive and used only for small caches.
- Cache Direct Mapped, which is the opposite of the previous cache structure. In fact, in this case, each location in main memory can be mapped in just one cache entry. Hence, searching for a given data is a cheap process since a single location needs to be checked. Thus, such structure is very convenient for large caches. However, the fact that each data has a predetermined location leads to cache conflicts where few

data contend for the same cache entry despite large parts of the cache are empty.

- N-way set associative cache is a combination of direct-mapped and fully-associative caches. Each address is placed to a predetermined cache set, as in direct-mapped caches, but in each set there are multiple entries (the same number in each set) and lines can be freely allocated in any line within their set, as in fully-associative caches. Hence, the degree of associativity (number of entries per set) determines the performance and efficiency of these caches. In general, they are the preferred choice for large caches since they allow obtaining most of the benefits of fully-associative caches with costs close to those of direct-mapped ones.

Note that N-way set associative cache is the cache structure that is mainly implemented in almost all the main caches of ARM cores [7]. In fact, in this Master thesis all the level caches of the two ARM architecture processors part of the Zynq board are N-way set associative caches. For this reason, further details are given about such cache structure.

N-way set associative cache

An N-way set associative cache structure is conceptually arranged into S sets (rows) and N columns (ways), as shown in figure (2.2). Each cell in the plot is a cache line. Each location in main memory can be mapped to one and only one set, but its contents can be placed in any of the cache lines (ways) in that set. Therefore, the lookup of a specific data is made in a group of N cache lines (those within the corresponding set).
Cache lines have a specific size (in bytes). In general, for the sake of implementation efficiency, all parameters are powers-of-two, and the size of the cache is determined as the product of the number of ways (N), the number of sets (S) and the cache line size (B). For instance, a 4-way cache with 128 sets and 64-byte cache lines is a 32 kB cache.

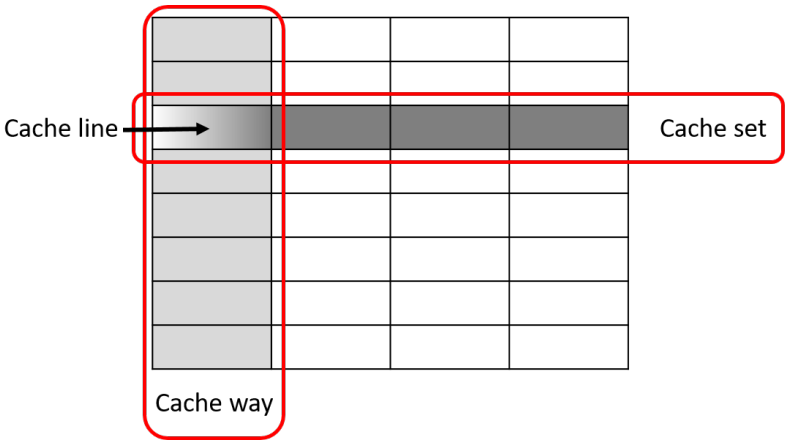

Figure 2.2: 4-way set associative cache

2.2.2 Cache policies

Different policies are needed in caches regarding replacement of lines on full sets, write policies, inclusion policies, etc. It is not the purpose of this section reviewing all those policies, but providing insights on those that are relevant for this Master thesis, so next we introduce some Replacement policies and Write policies.
Focusing on the first ones, there are two of them that are quite popular and employed in many caches:

- Replacement policies:
  - Least-Recently-Used (LRU), which aims to replace the data in a cache line that is the least recently used out of all those in the set. This policy builds on the fact that it is quite common reusing data recently used.
  - Pseudo-Random, which ensures that on a miss, a way in the corresponding set is randomly evicted to make room for the new cache line [8]. As this policy is used in some of the caches of the platform considered in this thesis, we analyze it later in detail.
- Write policies:

- Write-back (WB), which updates the main memory just when a cache line is evicted in the cache [7]. Thus, on a write operation, if the data is present in cache, it is only updated in the cache. This policy is quite convenient in terms of performance, since many memory accesses are avoided, but it's complex in terms of implementation since delayed memory writes need to be managed.
- Write-through (WT), which aims to update both the cache and the main memory of the system upon a write operation [7]. Note that this policy is not so convenient in terms of performance, since each write operation is forwarded to memory, but on the contrary it's easier to implement it in the target cache since there is no need to control dirty lines and perform delayed updates.

Pseudo-Random Replacement policy

Knowing that in a N-way set associative cache it's implemented a Random Replacement policy, it's possible to note that the probability that a specific cache line will be evicted is equal to $\frac{1}{N}$ for each set [8]. Hence, cache hits or misses are, in theory, truly probabilistic within the cache set. It has been shown that the hit probability for a specific access, for instance A_j, in an access sequence to its cache set like $< A_i, B_{i+1}, ..., B_{j-1}, A_j >$ is obtained using the following equation:

$$P_{hit_{A_j}} = \left(\frac{N-1}{N}\right)^{\sum_{k=i+1}^{n-1} P_{miss_{B_k}}} \tag{2.1}$$

where B_k corresponds to the accesses that are performed to cache lines different from the one where is present$A$.
Therefore, the probability that A is not evicted upon an eviction is equal to $\frac{N-1}{N}$, meaning that increasing the number of evictions in the cache set, increases the probability to evict A as well [8].
While this is the theoretical behavior of random replacement policies, actual implementations in processors may use poor pseudo-random number gener-

ators that do not produce fully random replacements. As shown later in the evaluation section, results show that, in fact, the random replacement policies implemented in the target platform may not be sufficiently random.

2.3 SPARC instruction set

In order to devise correctly the microbenchmarks for the ARM processor architectures, firstly the microbenchmarks employed for the Next Generation MicroProcessor (NGMP) were studied. Note that this architecture implements a SPARC V8 quad-core processor, which was developed by Cobham Gaisler for the future European Space Agency (ESA) missions [6]. For this reason, it was needed to study some instructions provided by the SPARC V8 instruction set, including the syntax implemented in such processor architecture in assembly programming language.
The most important instructions to be studied are the ones relative to memory write and read operations, since they need to be reproduced in the ARM instruction set to produce analogous access patterns:

- `ld` stands for *load*, which has the following syntax:

$$ld\ [r_s],\ r_d \tag{2.2}$$

  where r_s is the *source* register and r_d is the *destination* one.
  This is a memory read operation and it fetches from the main memory the data that is stored in the memory address specified in the register r_s. Afterwards, the content fetched is saved in the register r_d of the SPARC processor [9].

- `st` stands for *store*, which has the following syntax:

$$st\ r_s,\ [r_d + offset] \tag{2.3}$$

where it's employed the same notation used for equation (2.2).
The store instruction is a memory write operation, which has the goal to deliver into a specific memory address defined in the register r_d of the main memory a specific data that is found in the register r_s [9]. Note that between the brackets where the register r_d is placed, there is the possibility to add an *offset* value, which is a number that allows to deliver the data in memory addresses close to the initial one. This feature is very useful for the microbenchmarks to perform many store operations in memory addresses that are close one another.

Knowing the syntax of these instructions, it was easier to study the syntax of ARM processor architectures studied for the Zynq UltraScale+ platform, which is different from the one implemented in the SPARC one.

2.4 Multi-cores: benefits and drawbacks

Before going into deep details of the platform studied for this work, it's important to recall issues that lead to the use of multicore system and the tradeoffs involved in their implementation.
The best choice in terms of complexity and efficiency is to employ cores that are devised exclusively to perform the specific tasks they are intended to execute. However, in general, real systems end up executing a large variety of tasks and hence, many commercial multicores use general-purpose processing cores. However, even in this context, those cores may be especialized to some extent so that some cores prioritize performance over power or vice versa, or limit complexity, etc. Further, those heterogeneous cores can be deployed together in the same platform [10] so that end users (or some software layers on their behalf) can offload their applications on those cores that are expected to maximize the metric of interest (e.g. performance, power).
Multicores replicate across cores those resources with higher stress to increase performance, whereas those resources with a typically lower utilization are shared across cores for the sake of efficiency. For instance, the utilization

of some large cache memories and memory bandwidth is relatively low for many applications. Hence, it is common setting up processors with multiple cores that, beyond a given level of the cache hierarchy, share the rest of the hierarchy (e.g. L2 cache and main memory access channels).
Of course, sharing some resources, despite being an efficient solution in terms of resource utilization, bring a new issue: access arbitration due to contention. This is a relevant challenge since multiple cores may attempt to access a given shared resource simultaneously, and arbitration policies must provide balanced choices not to starve any core or, at least, configurable arbitration so that the user can decide what the most convenient way to share the resource is. Often, policies like round-robin are used to grant access to shared resources, so that all cores are granted access to the shared resource periodically. Still, if the amount of requests to access this shared resource is high (at least during some time periods), requests may get delayed, thus leading to lower performance to that that would be obtained on a single core architecture.

2.5 Zynq Ultrascale+

The Xilinx® UltraScale multiprocessor system-on-chip (MPSoC) considered for this Master Thesis implements in the same device both, a processing system (PS) and user-programmable logic (PL).
For what concerns the PS, it features three main processing units.

- Cortex-A53 application processing unit (APU)
- Cortex-R5 real-time processing unit (RPU)
- Mali-400 graphics processing unit (GPU)

The first two individual embedded blocks are the ones targeting general-purpose applications, and those of interest for the work in this master thesis.

In particular, the Zynq UltraScale+ platform includes two clusters of processors that feature two different architectures: the ARM v8 architecture-based 64-bit for the APU and the ARM v7 architecture-based 32-bit for the RPU.
In the following subsections, Cortex-A53 and Cortex-R5 cache features will be described in detail.

2.5.1 ARM Cortex-A53 Processor

The APU of this platform consists of four Cortex-A53 MPCore processor cores and a L2 Cache, which is a shared resource among these four processor cores.
The Cortex-A53 processor is devised with a modified Harvard architecture that leads to have different buses both for instructions and data. For this reason, in the internal Level-1 (L1) cache, there are two caches, i.e. instruction cache (I-cache) and data cache (D-cache). Moreover, L1 Caches are implemented as Set associative caches.
Next we describe the main parameters of the APU with regard to its cache hierarchy:

- ARM v8-A architecture instruction set.
  - Possibility to choose either A64 instruction set in 64-bit mode or A32/T32 instruction set in 32-bit mode.
- I-Cache and D-Cache are separated.
- Cache size of both L1 caches corresponds to 32 KB.
- Cache line size is fixed to 16 words, which corresponds to 64 bytes, both for L1 I-Cache and L1 D-Cache. Hence, each of those caches has 512 cache lines of 64 bytes each.
- L1 I-Cache is implemented as a 2-way Set associative cache, whereas the L1 D-Cache is implemented as 4-way Set associative cache. Hence,

the L1 I-cache has 256 sets with 2 cache lines each, whereas the D-cache has 128 sets with 4 cache lines each.

- Level-2 (L2) Cache size is equal to 1 MB.
- The replacement policy implemented for L1 caches is the Pseudo-random one.
- For what concerns the cache update policies, the L1 data and L2 caches use write-back policy. Since the I-cache does not modify the code stored, it does not need any write policy.

2.5.2 ARM Cortex-R5 Processor

The RPU is a cluster including a dual-core Cortex-R5 for real-time processing.
It's important to note that also in this case a modified Harvard structure and N-way Set associative caches are implemented in the Cortex-R5 processor cores.
The main features of those processors are reported below:

- ARM v7-R architecture instruction set.
  - The available instruction set is A32/T32.
- Instruction and data caches are separated thanks to the implemented Harvard architecture.
- Cache size of both L1 caches corresponds to 32 KB.
- Cache line size is fixed to 32 bytes, which corresponds to 8 words of 4 bytes each, both for instruction and data caches. Hence, each cache has 1,024 cache lines.
- L1 instruction and data caches are 4-way Set associative. Hence, they have 256 sets with 4 cache lines each.

- Level-2 (L2) cache is not present in this cluster.
- Caches of the Cortex-R5 cores implement Pseudo-random replacement policy, which is the same implemented in Cortex-A53.
- The write-back policy is implemented in the Cortex-R5 L1 data cache.

Chapter 3

State of the art

As explained before, it is common in industry relying on MBTA for WCET estimation, and some approaches based on the use of microbenchmarks to model multicore contention have been found appropriate. Hence, in this section we review some of the main works in the are of microbenchmark development to induce high stress conditions in multicores. In particular, approaches generating stressful scenarios consider not only performance stressful conditions, but also power and temperature conditions as a means to assess relevant non-functional metrics of processors and applications.

3.1 Performance Stressing Benchmarks

In the context of critical real-time systems, and with particular emphasis on commercial off-the-shelf (COTS) multicore processors, software testing has been largely exploited to test functional and non-functional properties of software. In particular, those tests are run during the analysis phase of the system, in early design stages, when many applications are still under development. In the case of WCET estimation, the objective is obtaining WCET estimates during unit testing (i.e. when the task under analysis has been implemented), without the need of waiting for other units (e.g. tasks

that will run concurrently) to also be implemented. This allows assessing whether execution time budgets allocated to tasks suffice to run them and, if this is not the case, address this issue as soon as possible, since detecting timing violations during late design stages incurs high costs and may impact time-to-market.

Therefore, since tasks running concurrently are unknown during WCET estimation, assumptions need to be made on the contention those tasks can generate. Usually, this has been accounted for using simple programs (aka microbenchmarks) that place specific amounts of contention on specific shared hardware resources. For instance, one may develop a microbenchmark reading constantly from memory to generate high contention in the access to memory to measure how sensitive the task under analysis is to such contention.

Strategies to create contention relevant for WCET estimation are diverse. Some authors created those types of microbenchmarks to study the impact of contention on high-performance Intel and AMD processors [11]. While those processors are generally regarded as inappropriate for critical real-time embedded systems due to the large number of hard-to-control sources of execution time variation, the strategy followed to develop microbenchmarks has been later reused to evaluate more appropriate processors. In particular, authors developed microbenchmarks with simple loops sufficiently small to fit in L1 instruction caches, but sufficiently large so that the overhead to increase the loop counter and jump was negligible. Then, those loops contain a sequence of instructions of the same type accessing a especific shared resource (e.g. the second level, L2, cache) with the aim of creating the highest contention possible.

A similar strategy was applied on the Cobham Gaisler LEON4 processor [6], whose target is the Space domain. Experiments performed with microbenchmarks revealed that an early design of the LEON4 allowed to cause 20x slowdowns on a 4-core multicore. Microbenchmarks showed that, by using a non-split bus, the worst contention impact occurred when the task under

analysis was performing sustained L2 hits whereas all other contenders were performing L2 misses. Upon an access to L2, the non-split bus gets locked by the accessing task and it is not released until the transaction completes. Hence, each (short) L2 hit of the task under analysis may have to wait for 3 memory accesses (L2 misses) caused by each of the 3 contender tasks. This behavior turned to be particularly exacerbated for store operations whose latency for L2 hits is very low, whereas sustained store L2 misses caused 2 memory accesses each: one to evict a dirty line modified by previous stores and another to fetch the line accessed by the store itself.

While maximum stress contention scenarios are relevant for WCET estimation, they may be overly pessimistic for some platforms and applications. Hence, some authors extended microbenchmarks for the LEON4 platform to consider especific amounts of contention [12]. Authors build upon the concept of partial time composability instead of full time composability, meaning that contention bounds obtained are only valid under especific amounts of contention. In particular, authors show how to account for especific access counts to each shared hardware resource so that contention bounds obtained are valid as long as contenders do not exceed those access count bounds. This approach is particularly useful when some information about contenders is available, so that especific access count bounds can be set with the aim of upper bounding real access counts but without having to account for the maximum number of accesses possible. This approach builds on the use of maximum stress microbenchmarks to derive per-resource latencies, which are later used to statically model the maximum contention possible under especific loads. Also, this work devises microbenchmarks performing especific access counts as a means to verify that contention bounds estimated statically match those obtained empirically in the worst case.

A similar approach has been followed for the Infineon AURIX TC27x processor family [13]. Due to the particular characteristics of this platform to count events, an Integer Linear Programming (ILP) model has been developed to obtain upper and lower bounds to access counts from stall cycle

counters. On the other hand, microbenchmark technology to measure maximum latencies and to create contention scenarios is the same as for [12].

Such strategy has also been considered for the Qualcomm SnapDragon 810 processor within the framework of the H2020 SAFURE project [14]. Such processor has been regarded as appropriate for the telecommunications domain and used in many embedded systems such as the Sony Xperia smartphone. While the strategy followed for this platform has been analogous to that for the LEON4 and AURIX processors, results showed that documentation is incomplete and inaccurate for this platform [15]. In particular, the prefetcher could not be disabled and events monitored by PMCs were insufficient to estimate contention with meaningful accuracy. Thus, this platform has been regarded as inappropriate unless documentation available increases so that the platform can be mastered to a sufficient extent.

Other authors attempted to model the contention in the interconnection network of the NXP P4080 processor – relevant for avionics and railway domains – by developing similar microbenchmarks [16, 17]. Their work revealed that contention is not linear with the number of cores, thus exposing that, while this 8-core architecture may seem to be symmetric, it is not. In particular, experiments revealed that contention caused by some cores was higher than that caused by others, thus exposing the fact that the interconnection network organizes cores into two different clusters, and contention between cores of the same cluster may be higher than across clusters. Further analysis of the NXP P4080 has been carried out with microbenchmarks assessing other types of execution time interference across cores, and revealing that, for instance, some asymmetric behavior is caused by snoop accesses for cache coherence despite tasks run may not share any data [18].

Other approaches have focused on performing some form of stochastic analysis of contention with the aim of identifying typical timing behavior under high contention on shared hardware resources, but without explicitly considering the worst case, thus providing a family of testing techniques building on the correlation of PMCs [19]. Those approaches build also upon the use

of microbenchmarks to expose dependencies across events which, ultimately, requires the creation of some microbenchmarks producing high contention to reveal dependencies in the access to shared hardware resources. A similar approach building on similar types of microbenchmarks have been devised with the aim of applying statistical techniques such as principal component analysis to predict the worst contention bounds of critical real-time tasks on multicores [20].

Finally, some authors have attempted to model contention at late design stages by running simultaneously tasks that may contend against each other, modifying their time alignment (i.e. their relative starting time) to account for the worst – yet realistic – contention that especific tasks can cause on each other [21].

3.2 Power and Thermal Stressing Benchmarks

Stressing benchmarks have been used in other contexts with the aim of predicting other non-functional metrics such as power and temperature. Next we provide few illustrative examples of those applications of microbenchmarks.

Thermal analysis by means of software-based solutions has been mostly considered for post-silicon validation of processors with the aim of identifying the Thermal Design Power (TDP), which is the maximum sustained temperature that a processor can produce. A proper identification of the TDP is key for chip manufacturers to size the cooling solution needed to keep the processor under specific temperatures. Triggering the TDP typically requires the development of the so-called power virus programs, which create sustained high-power activities [22, 23]. For instance, floating point operations have been shown to consume high energy, while allowing virtually executing one such operation per cycle per core. Hence, microbenchmarks building on those types of operations are often used to trigger specific high-temperature scenarios.

In essence, benchmarks intended to trigger high execution times (due to contention) or high temperature have similar structures (loops with specific patterns that repeat many times) varying the type of operation that is executed sustainedly depending on whether the objective is to create high contention or high temperature.

Chapter 4

Methodology

In this chapter, the methodology used for the experiments performed will be explained in detail, explaining why and how such experiments are performed in order to achieve the expected objectives. In particular, different microbenchmarks and functions devised both in C/C++ and assembly programming languages will be addressed in details, describing how they work and their features.
In this thesis, the methodology consists of using specifically designed microbenchmarks to collect empirical evidence directly on the target platform. Those microbenchmarks are executed in isolation in some experiments and together with other microbenchmarks (either identical or different) in other cores in other experiments. Results of the execution are collected reading the PMCs that exist in the platform under study. In order to ensure that results are reliable, the following steps have been followed, which attempt to expose sources of variability incrementally:

- *Tests on one core of the Cortex A53 and Cortex R5 cluster processors of each relevant microbenchmark devised.* These tests have to be performed when all the cores (except the one where the microbenchmark runs) are in power-down mode, avoiding any type of interference. In this way, it can be verified that both the microbenchmark and the tar-

get main core are working properly. Moreover, it is possible to verify if the features represented in the corresponding manuals are correct or not. The rationale behind these experiments is that microbenchmarks have a known behavior on the specific hardware where they are run. Thus, approximate values for some parameters such as executed instructions, cache accesses, memory accesses, etc. are known a priori and can be doublechecked against results measured.

- *Study of PMCs behavior*, understanding if they work properly and how much noise and other type of disturbances can affect their operation. In this way, it is possible to assess that such counters are reliable enough to be used for achieving the goal of this Master thesis. This is an important step since it is not unusual having documentation where PMC description is scarce, so that their definition is ambiguous. Also, since PMCs are not critical for operation, they are often less debugged that other parts of the processor and may have unexpected behavior (e.g. counting just a subset of the events they should).

- *Collecting data of the PMCs when all cores are active*, thus verifying that all PMCs can be interfaced properly even when all cores in both clusters are running. This is important to verify that events corresponding to shared hardware resources can be counted on a per-core basis, thus avoiding interference on PMC values themselves.

- *Run several experiments with all the cores in running mode, using the same microbenchmarks devised before.* Each experiment is distinguished among the other ones since that different microbenchmarks are executed in parallel, meaning that there are processor cores behaving as "contenders", which are the processor cores that can generate inter-task interference due to the contentions that arise in the hardware shared resources with the main core under observation. These are the most relevant experiments since they represent the scenario that a given critical real-time task may experience in the system during operation.

- *Collecting data of the PMCs results when all the processors are executing simultaneously*, comparing them with the ones obtained for single cores. While the previous set of experiments exposes the actual behavior in terms of execution time, this set of experiments offers details on why execution time in parallel operation differs from that in single core operation. Thus, results from this set of experiments allows reaching conclusions on what type and degree of interference occurs in each shared hardware resource. Such information is crucial to understand what the most convenient way to consolidate tasks is.

4.1 Microbenchmarks

As explained before, microbenchmarks are becoming more and more useful to evaluate performance of multi-core architectures. Reasons of such statement are confirmed from the fact that they are designed to be easily adapted to other processor architectures thanks to the simplicity with which they can be developed in programming languages like C/C++. Also, industry preference for quantitative evidence on the target platform is also a plus for this approach.
The size of each microbenchmark (in terms of code footprint) is small enough to fit in the instruction cache. This allows controlling the stress on each shared hardware resource by controlling the amount of data used, the access frequency and the type of access, without suffering any meaningful interference from the code itself. Therefore, the experiments are focused mainly on generating very high loads in the different levels of the memory hierarchy, namely L1 data and the L2 cache, and main memory.
Algorithm 1 represents the conceptual schematic employed for the microbenchmarks implemented in the target platform studied in this Master thesis.
We identify each algorithm with appropriate names (i.e. `Name_Microbenchmark` in the algorithm). Then, the parameters `A` and

`B` represent the input arguments of such functions written in C/C++ programming language, where `A` changes depending on the type of operations performed, while `B` is the number of times that the main loop of the microbenchmark has to be executed. Afterwards, memory allocation and initialization procedures take place. Note that memory initialization may require setting specific contents in memory so that the main loop of the microbenchmark maximizes the number of accesses per cycle as detailed in next chapter.
Finally, the loop represented in such algorithm is written mostly with the ARM instruction set (assembly code) and it is featured by *Memory instructions* that are repeated several times. Note that the type of instruction chosen and the number of instructions defined change among the microbenchmarks depending on what effect they are intended to produce. Therefore, there are memory instructions that read from the main memory or cache (loads) and other ones that write to the main memory or caches (stores).
As shown in the next chapter, for implementation efficiency and flexibility purposes, the initialization/allocation phase and the main loop can be decoupled across different functions so that initialization/allocation code is shared across different microbenchmark types.

4.2 Performance Monitoring Counters

The quantitative evaluation of the results obtained with the microbenchmarks described in the previous section (4.1) is performed through the use of the *Performance Monitoring Units* (PMUs) existing in each core, which include a set of *Performance Monitoring Counters* (PMCs) each.
These units are included in both the Cortex-A53 and Cortex-R5 processor cores and they are useful in order to verify that the experimental results correspond to the expected ones, and to analyze how interference occurs in shared hardware resources.

Algorithm 1 General structure of the implemented microbenchmarks

```
int Name_Microbenchmark (A, B)
{
  /* Start memory allocation procedures
      (...)
  */ End of such operations

/* Start assembly code
   Start Loop
     Memory Read/Write Instruction (A' -> Register/Main Memory)
     Memory Read/Write Instruction (A' -> Register/Main Memory)
           (...)
     Memory Read/Write Instruction (A' -> Register/Main Memory)
   End Loop
*/ End assembly code
}
```

In fact, the PMCs can be configured to measure different events, so they need to be configured properly to count the events of interest. Then, it is important to enable PMCs right before the execution of the microbenchmark and disable them right after so that only the activity of the microbenchmark is effectively monitored. Once this is guaranteed, the values obtained for a given event type with PMCs can be compared against the values expected. If the number obtained for one event is the expected one, then we can rely on the PMCs counting such event correctly, so that knowledge can be built on top of its results when collecting data for single core and multi-core experiments. For this reason, firstly experiments were performed in order to check if the PMCs of such processor are working properly. Later, after that PMCs are confirmed to work properly, they were interfaced in C/C++ programming language together with the microbenchmarks. The PMCs interfacing has been integrated in the function where the microbenchmark was defined in order to improve the reliability of the measurements of each event counted. The PMU events available are several for these architectures and in the fol-

lowing are listed the most important ones that are considered for the experiments performed. Firstly, the common events are reported and afterwards the Cortex-A53 and Cortex-R5 specific events are described.

4.2.1 Common events

1. L1D_CACHE - This counter stands for L1 Data cache access and it counts how many times the L1 Data cache is accessed by read and write operations. Therefore, read and write accesses are not discriminated.

2. L1I_CACHE - L1 instruction cache access, which counts instruction memory accesses to both the L1 Instruction cache and L1 instruction memory structures like refill buffers.

3. L1D_CACHE_REFILL - L1 Data cache refill, which corresponds to the number of read and/or write misses that occurs in the L1 Data cache. In this case, the PMC counts each access to L1 cache causing a refill of a cache line brought from the upper level (either main memory or another cache level).

4. L1I_CACHE_REFILL - L1 Instruction cache refill, which corresponds also to the number of read misses that occurs in the L1 Instruction cache. Note that instructions can only be read in general, so only read operations can occur.

5. CPU_CYCLES - This counter counts the number of processor cycles. Note that the processor may operate at a different frequency than other components (e.g. main memory), so the operating frequency of the processor needs to be used to obtain execution time. Combining this counter with other ones, it's possible to figure out, for instance, the frequency at which some specific events occur (e.g. number of L1 data cache misses per cycle).

4.2.2 Exclusive Cortex-A53 events

Since the architecture of the Cortex-A53 cluster differs from that of the Cortex-R5 cluster, some events are specific for each core type. For instance, the Cortex-A53 cluster has an L2 cache in between L1 caches and memory, whereas the Cortex-R5 cluster has not.

1. L2D_CACHE_REFILL - L2 Data cache refill is the event that counts the number of accesses to the L2 cache causing a refill of a L2 cache line, regardless of whether they also cause a refill of the L1 instruction cache, the L1 data cache or none of them. This means that events 4 and 3 may overlap with this event if they miss in L2. Such observation is backed in chapter (6), within which the results of this work are presented.

2. L1D_CACHE_WB - L1 Data cache Write-Back, which corresponds to the number of write-back of data performed from L1 Data cache to higher memory levels like L2 cache or the main memory. In other words, it counts how many times a modified line in L1 Data cache is evicted.

3. L2D_CACHE_WB - L2 Data cache Write-Back, which corresponds to the number of write-back of data performed from L2 Data cache to main memory.

4. L2D_CACHE - L2 Data cache access counter considers all accesses to a cache line of the L2 Data cache caused by read and write operations coming from the cores. Therefore, it includes the number of refills of both L1 Instruction and Data caches and the number of write-backs of data performed from L1 data cache. This means that this event is the sum of all the accesses performed to a cache line of the L2 cache performed by L1 caches (so events 4, 3 and 2).

5. APU_EXT_MEM_REQUESTS - The external memory request counter is defined for the APU of Cortex-A53 and it increments for each memory

access request that is external, thus including L2 cache misses and L2 cache write-back operations.

4.2.3 Exclusive Cortex-R5 events

1. RPU_DCACHE_WB - L1 Data cache Write-Back counter for the two processors in the RPU cluster increases when one write-back of data is performed from L1 Data cache to higher memory levels like the main memory. Note that for this processor there is no L2 data cache, but just Tightly Coupled Memories (TCMs) and the main memory.

2. RPU_EXT_MEM_REQUESTS - This PMC is defined for Cortex-R5 in the RPU cluster and it increments for each access request from L1 data cache to an external memory like the main memory. It is analogous to APU_EXT_MEM_REQUESTS, but for the Cortex-R5 cluster instead of for the Cortex-A53 one.

4.3 Tools

The following tools were employed for this Master thesis:

- Debugger interface for Zynq UltraScale+ EG platform, which is called Xilinx System Debugger, and it is exploited for directly connecting to the board, in order to perform operations like reading the registers of each processor core in real-time or reset the target processor when errors occur, among other operations.

- MobaXterm, which is a software for Windows for remote access to other computers. This software allowed to connect to the private BSC server, within which Unix commands have to be used in order to compile codes and to run experiments. Such BSC server is the host of the Zynq platform and has a cross-compiler able to generate ARM binaries to be run on the Zynq platform.

Chapter 5

Implementation

This chapter explains in details how the microbenchmarks described theoretically in section 4.1 are implemented in practice through the use of both C/C++ and assembly programming languages. Firstly, insights of the main function are given in section 5.1 and descriptions and features of the experiments performed for this study that are declared inside the aforementioned main function are listed in section 5.2.

5.1 Main function

The experiments were performed using the main function represented in Algorithm 2. Note that the overall execution time is spent in the *Array initialization* function (executed just once) and the microbenchmark chosen to be run. In particular, the latter is nested in the `do/while` loop, which can be an infinite loop if the `status` variable returned by the microbenchmark is always 0. In this way, such microbenchmark can be executed an infinite number of times, stressing a specific resource of the target processor sustainedly as desired, and accounting for an execution time arbitrarily larger than the initialization part.
Note that it was needed to define some C/C++ pre-processor commands in

order to make the code more flexible. In fact, as shown in the main function, depending on whether the *macro-name* `A` or `B` are defined or not, it is possible to initialize the array in different ways. Other pre-processor commands were implemented, which are not shown in the aforementioned function just for the sake of simplicity.

Algorithm 2 Main function

```
{
    int status = 0;
    Disable_Prefetch();

#if defined A    // Array initialization function...
    int **array = mem_init(array_size, stride);
#elif defined B     //...to access the same set.
  int replacements = X;     // N° of replacements;
  int **array = mem_init(One-Way_stride*replacements, One-
    Way_stride);
#endif

  do
    {
        status = Microbenchmark;  // Desired microbenchmark
    }
    while (status==0);

    return 0;
}
```

5.2 Experiments

The experiments performed to stress the cache hierarchy of the two different processor clusters are the following ones:

1. L1 data cache and L2 cache read misses and hits accessing different sets.

2. L1 data cache and L2 cache read misses and hits forcing the processor to access always the same set.

3. L1 data cache and L2 cache store misses and hits.

In the following sections the microbenchmarks used for such experiments written in C/C++ and assembly programming language will be presented in details. In particular, section 5.2.1 introduces read microbenchmarks, section 5.2.2 introduces write microbenchmarks, section 5.2.3 describes how prefetch interference is avoided, and section 5.2.4 describes how PMCs have been interfaced.

5.2.1 Cache read operations: Load instructions

The microbenchmark devised for cache read operations is used for both L1 and L2 data cache. In fact, setting properly both the size of the whole array and the stride for each array element, it is possible to impose cache read hits or misses in L1 or L2 data cache. Since updating the stride to access the following element requires at least a non-load operation, this could lead to a load frequency lower than the maximum possible. To address this concern, it is needed to implement the Pointer Chasing technique (Algorithm 3). Such pointer chaising is part of the initialization process and hence, needs to be performed before the actual microbenchmark code (e.g. Algorithms 4 and 5) is executed. Next, we introduce how pointer chaising works and how it allows read microbenchmarks execute roughly only load operations.

Array initialization using pointer chasing

The pointer chaising approach aims at placing in memory the addresses of the data to be loaded in the location loaded right before so that, on a load, we fetch the (precomputed) address of the next address to be loaded, so that we avoid having to compute such an address during the execution of the microbenchmark.

As shown in the Array initialization code (Algorithm 3), the *mem_init* function receives two inputs: the size of the array (array_size) and the distance between two consecutive array elements (stride). Afterwards, their initial value is divided by the size that the pointer to an integer variable occupies in the memory of the processor, which changes from architecture to architecture. This step is needed since each array element occupies 8 bytes in memory but the actual size of the pointer may vary across architectures, being either 4 or 8 bytes. Hence, we must make sure that we access appropriate addresses and the specified number of times. If this step was not done, there would be the risk that, after a number of accesses, memory accesses could occur beyond the boundaries of the array, thus leading to potential memory violations and, more importantly, to undesired timing behaviour for the microbenchmark.

Afterwards, we allocate the required amount of memory for the microbenchmark with the malloc function given by C/C++ programming language. Later, the real array initialization takes place, which is mainly achieved using a "for" loop with a simple conditional inside to capture the case of the last element of list. Therefore, the first array element will store the memory address of the next array element that is placed in `cnt+stride`, which corresponds to the stride updated on each iteration. The stride is chosen in order to not violate the word-alignment, meaning that it has to be a multiple of 4 bytes. Note that `cnt` variable increases in each iteration by an amount equal to the stride defined initially. This means that just some array elements will be accessed by the microbenchmark, while the other ones will remain unset. For instance, with a stride of 16-bytes, the first element of the array contains the address of the array element 16 bytes away, which in turn contains the address of the array element 16 further bytes away, and so on and so forth. Elements in-between those ones are neither set nor used.

The microbenchmark algorithm uses this initialized array in order to perform cache and memory read operations.

Algorithm 3 Array initialization using pointer chasing

```
int **mem_init(unsigned long int array_size, int stride)
{
    array_size = array_size / sizeof(int*);
    stride = stride / sizeof(int*);
    int**array = (int**) malloc(sizeof(int*)*array_size);

    unsigned long int cnt;
    for(cnt=0; cnt < array_size; cnt+=stride)
    {
        If(cnt < array_size - stride)
            {
              array[cnt] = (int*) &array[cnt+stride]; //Each array
      element points to the address of the next array element.
            }
        Else
           {
             array[cnt] = (int*) array; //The last accessed
     element in the array points to the first element.
           }
     }

     return array;
}
```

Microbenchmark based on Load instructions

The code aimed to perform memory read operations (Algorithm 4) is based on Load instructions, which are defined in the processor architecture and they have to be written for this reason in assembly code to ensure that the compiler does not alter the access pattern or decreases the load frequency. Note that this corresponds to the first experiment (1) performed for the two different clusters.

Firstly, two pointer variables are defined in order that they will point to the same address of the first element of the initialized array described previously

(Algorithm 3). Afterwards, the first load instruction (`LDR`) is performed, which fetches the content of the first array element from the main memory and puts it in a register of the processor architecture.
Since the content of the first array element corresponds to the address of the next array element to be accessed by this microbenchmark, the second load instruction fetches the content of the element whose address has just been fetched from main memory, putting it in another register. The same reasoning holds for the other 126 load instructions, where data loaded from memory is always the next address to be accessed, thus not needing to compute any address during the execution of the microbenchmark.
After these 128 load instructions are executed, a compare instruction (`CMP`) checks if the last memory address stored in the register accessed by the last load instruction corresponds to the one of the first array element. Depending on whether all array elements were accessed or not, and whether all iterations have been exhausted, either we iterate in the loop performing further load instructions or the loop finishes. Such control is performed with the `BNE` and `SUBS` instructions at the end of the loop.
Note that this microbenchmark is used for causing both, either all cache hits or all cache misses. In fact, such events occur depending on the size of the whole array to be stored in the target data cache and the particular stride used. Therefore, if the whole array fits completely in the target data cache, cache hits occur, otherwise cache misses will be experienced as long as the stride is equal or larger than cache line size (under Least Recently Used, LRU, replacement policy). Other cases are not relevant for contention evaluation since we look for pure-hit or pure-miss cases in each cache memory, so mixed behaviour is not interesting.
For instance, with this behaviour we can generate a microbenchmark hitting in L1 (array size not exceeding L1), missing in L1 and hitting in L2 (array size larger than L1 but not exceeding L2 size), and missing in both L1 and L2 (array size larger than L2). In all cases, we use a stride matching a cache line size to ensure that accesses occur in different cache lines so that cache

line locality does not interfere with the experiment.

Algorithm 4 Microbenchmark based on Load instructions

```
int Loads(register int** array, register int iterations)
{
    register int** q = array;
    register int** r = array; //Array provided by the Pointer
    Chasing algorithm.

/* Start assembly code
    __asm__ __volatile__(
    ".data_cache_label_L1:"   "\n\t"
    "LDR %0, [%1]" "\n\t"
    "LDR %1, [%0]" "\n\t"
                ...
                ...
    "LDR %0, [%1]" "\n\t"
    "LDR %1, [%0]" "\n\t"  //Total of 128 load instructions
    "" "\n\t"
    "CMP %2, %1"   "\n\t"
    "BNE .data_cache_label_L1" "\n\t"
    "SUBS %3, %3, \#1" "\n\t"
    "BNE .data_cache_label_L1" "\n\t"
    ".label_L1_exit:"       "\n\t"
    :
    : "r"(q), "r"(array), "r"(r), "r"(iterations)
    );
*/ End assembly code

    return 0;
}
```

Microbenchmark for accessing the same set

The microbenchmark methodology has been assessed in the past on caches implementing LRU replacement, but not specifically on caches using pseudo-random replacement. To assess the impact of using such a replacement policy of both the ARM Cortex-A53 and the ARM Cortex-R5 processors, we needed to perform a second experiment (2) and to devise another microbenchmark aiming to access the same set of cache lines in the L1 data cache. For this reason, in the Array initialization code (Algorithm 3), the stride and the size have to be changed properly with respect to the ones employed in (Algorithm 4). In particular, the stride is changed making it match the size of one way of the L1 data cache, which is computed as follows, where the size of the cache and the size of the stride are expressed in the same units (i.e. either bytes or array elements):

$$One\ Way\ stride = \frac{Data\ Cache\ Size}{N^{\circ}\ ways} \tag{5.1}$$

Therefore, the stride is set in such a way that the next array element accessed is at 1-way distance in the array, so that it is mapped exactly in the same L1 cache set. The corresponding code is shown in Algorithm 5.
It can be noted that such microbenchmark is very similar to the one based on Load instructions (Algorithm 4). The main difference is the number of Load instructions employed. In fact, the number of ways in which the two levels data caches (L1 and L2) are divided for both processor clusters is lower than the 128 load instructions used in the Load instructions microbenchmark (Algorithm 4). For this reason, it is not needed to use so many Load instructions.
Moreover, by using a variable number of load instructions, the microbenchmark is made more flexible from a programming point of view allowing a finer-grain control on the number of different cache lines competing for the space in a cache set. In fact, the size of the whole array is set to be:

$$Data\ Cache\ Size = One\ Way\ stride \cdot N^{\circ}\ replacements \qquad (5.2)$$

where N° *replacements* stands for the number of different cache lines (number of replacements) competing for the space in a cache set. Hence, depending on the number of replacements chosen, the size of the array changes. According to Equation 5.2, the number of replacements corresponds to the total number of unique addresses loaded in the loop. Therefore, the real number of replacements performed in the same set of L1 (or L2) data cache has to be computed after all those addresses have been accessed for the first time so that a steady state is achieved.

Algorithm 5 Microbenchmark to access the same set

```
int LDR_L1waysN(register int** BDA, const int iterations)
{
  register int** p = BDA;
  register int ** q = BDA;

/* Start assembly code
__asm__ __volatile__(
    ".l1waysnloop_begin:" "\n\t"
    "LDR %1, [%0]"    "\n\t"
    "LDR %0, [%1]"    "\n\t"
    ""   "\n\t"
    "CMP %2, %0"   "\n\t"
    "BNE .l1waysnloop_begin" "\n\t"
    "SUBS %3, %3, #1"          "\n\t"
    "BNE .l1waysnloop_begin" "\n\t"
    :
    : "r"(p), "r"(BDA), "r"(q), "r"(iterations)
    );
*/ End assembly code

  return 0;
}
```

5.2.2 Cache write operations: Store instructions

For what concerns the cache and memory write operations, a microbenchmark different from the ones addressed in section 5.2.1 has to be developed. In fact, it is not needed anymore to create an initialized array variable, meaning that the Pointer Chasing technique (Algorithm 3) will not be employed for the third topology of experiments (3) shown in Section 5.2. The reason is that write operations will not be fetching data where we can have the pointer to the next address to access. Instead, contents will be sent to memory, so addresses to be accessed need either being computed or read from somewhere which, in practice, implies using non-store instructions to set the address to

be accessed by the following store instruction.
In the following, microbenchmarks that stress data write features in the L1 and L2 data cache are presented.

Microbenchmark based on Store instructions - Hits

The algorithm addressed in this section (Algorithm 6) employs Store instructions to perform memory write operations and its goal is to cause hits in the target cache of the Cortex A53 processor cluster. Such instructions, as for the Load ones, are pre-defined in the processor architecture and assembly language is needed to use them, avoiding the compiler to interfere with the desired microbenchmark behaviour.
Firstly, memory allocation is performed with the `malloc` function so that stores can be performed on this memory structure. The address of the allocated memory is stored in the `st_array` integer pointer variable within which the data will be stored. Afterwards, the `st_pointer` integer pointer variable is used to access the allocated memory structure, so it is defined to point to the memory address previously assigned to the `st_array` first. Then, the `useless_data` (DEADBEEF) is the data that will be stored in the allocated memory region. Note that this data corresponds to a size of 32-bit, namely one word size. This is in line with the fact that the stride between each Store instruction has to be a multiple of 4 bytes to avoid unaligned accesses.
The main core of this microbenchmark consists of a for cycle including a total of 32 Store instructions. Therefore, in each iteration, 32 memory write operations will be carried out, meaning that the word "DEADBEEF" will be written 32 times. Depending on the `iterations` variable value, it performs a higher or lower number of Store instructions and such value must be chosen so that the amount of data accessed (and how many times it is accessed) produces the desired behavior, which in this case is a sequence of store hits (except for the first access to each cache line).
The microbenchmark based on store hits represented in this section uses a stride of 64 bytes between each element, meaning that the `useless_data` will

be stored to memory addresses with 64 bytes of distance each other. Such stride is used for Cortex-A53 processor and it was chosen in such a way that each store instruction will access the next cache line (64-byte size for Cortex-A53 cores).
Each store instruction has the same memory address as the initial argument between square brackets plus an offset (stride) of 64 additional bytes with respect to the previous store. In order to allow that consecutive instructions are stores without needing instructions to update the stride, strides need to be written manually directly in the assembly code.
When the 32 store instructions are performed, an `ADD` instruction is used to update the total stride for the next iteration. In particular, such stride is 2048 bytes, thus equal to the 64 bytes per store multiplied by 32 store operations. In this way, in the next iteration, the store instructions will be performed on the next available memory address of the `st_array` variable.
Before the next iteration starts, it is checked whether the `st_pointer` variable points to a valid range of memory addresses, namely to the ones assigned to the array `st_array` by not exceeding the array size. Such array is sized to ensure that store instructions cause cache hits. In fact, in this work, 24 kB was defined as upper limit since the L1 cache size is equal to 32 kB. Therefore, after the first 12 iterations, which correspond to 24 kB, cache hits are always experienced due to the fact all the memory addresses are stored in the L1 data cache. Note that we could use up to 32 kB of data, but since some temporal variables are stored in cache, this would create some conflicts and so, some misses. Thus, we use an array size smaller than the total cache size.
Whenever the `st_pointer` can access beyond the bounds of the array, in order to avoid errors due to access non-allocated memory regions and also causing misses, it is set to the initial `st_array` memory address.
At the end of the microbenchmark, the function `free` is employed to clear the memory allocated to the `st_array` variable.
The reasoning made so far for Algorithm 6 also holds for the one represented

after this first one and used for the Cortex-R5 processor cores (Algorithm 7).
It's possible to observe that in the main differences between the two microbenchmarks are as follows:

- Stride value, which is of 32 bytes in Cortex R5 since cache lines in those cores are 32-byte instead of 64-byte long.
- The total number of instructions per iteration is set to 64 instead of 32 so that the amount of data accessed per iteration remains constant, although this constraint is not needed in practice and strides could be adapted accordingly.

Algorithm 6 Microbenchmark for store hits - Cortex A53

```
int store_hit(unsigned long int size, const int iterations)
{
    size = size / sizeof(int*);
    int *st_array = (int*) malloc (size*sizeof(int*));
    int *st_pointer = st_array;
    int useless_data = 0xdeadbeef;

    register int j;
    for (j = 0; j < iterations; j++) {

    /* Start assembly code
      __asm__ __volatile__(
        "STR %1, [%2, #64]" "\n\t"
        "STR %1, [%2, #128]"  "\n\t"
                ...
                ...
        "STR %1, [%2, #2048]" "\n\t" //Total of 32 stores
   instructions
        "ADD %0, %2, #2048"   "\n\t"
        : "=r"(st_pointer)
        : "r"(useless_data), "r"(st_pointer)
        );
    */ End assembly code

      if ((int)st_pointer+(32*64) > (int)st_array + (24*1024))
        st_pointer = st_array;
      else
        st_pointer += 0;
    }

  free(st_array);
  return 0;
}
```

Algorithm 7 Microbenchmark for store hits - Cortex R5

```
int store_hit(unsigned long int size, const int iterations)
{
    size = size / sizeof(int*);
    int *st_array = (int*) malloc (size*sizeof(int*));
    int *st_pointer = st_array;
    int useless_data = 0xdeadbeef;

    register int j;
    for (j = 0; j < iterations; j++) {

    /* Start assembly code
      __asm__ __volatile__(
        "STR %1, [%2, #32]" "\n\t"
        "STR %1, [%2, #64]" "\n\t"
                ...
                ...
        "STR %1, [%2, #2048]" "\n\t" //Total of 64 stores
    instructions
        "ADD %0, %2, #2048"    "\n\t"
        : "=r"(st_pointer)
        : "r"(useless_data), "r"(st_pointer)
        );
    */ End assembly code

      if ((int)st_pointer+(32*64) > (int)st_array + (24*1024))
        st_pointer = st_array;
      else
        st_pointer += 0;
    }

  free(st_array);
  return 0;
}
```

Microbenchmark based on Store instructions - Misses

The microbenchmark used to cause cache misses in Cortex A53 processor is represented below (Algorithm 8) and it is very similar to the one described previously (Algorithm 6), both in terms of concept and of implementation. The main difference is the upper limit of the data array traversed, which is `DCL2_SIZE*2` instead of 24 kB (so twice the size of the L2 cache) if accesses are intended to miss in L1 and L2. If accesses must miss in L1, but not in L2, then the size of the array traversed should be larger than L1 (32 kB) but smaller than L2 (1 MB). For instance, we could use an array of 64 kB. Therefore, each store instruction has to cause a cache miss, since each cache line is written once (at the beginning of every cache line) in the target data cache and, since the number of lines accessed is larger than the number of cache lines available (in each cache set), those cache lines cannot fit in the L1 data cache (and L2 cache).
An analogous microbenchmark is employed also for Cortex R5 processor (Algorithm 9) and the same observations made about Algorithm 6 are valid also for this one.
As before, the stride value is 32 bytes instead of 64 bytes. Also note that the upper limit of the array traversed is, in this case, `DC_SIZE*2`, where `DC_SIZE` corresponds to 32 kB, namely the L1 data cache size, since the Cortex-R5 processor cluster does not have L2 cache. Therefore, systematic cache misses are experienced since the amount of data accessed exceeds L1 cache space available.

Algorithm 8 Microbenchmark for store misses - Cortex A53

```
int store_miss(unsigned long int size, const int iterations)
{
    size = size / sizeof(int*);
    int *st_array = (int*) malloc (size*2*sizeof(int*));
    int *st_pointer = st_array;
    int useless_data = 0xdeadbeef;

    register int j;
    for (j = 0; j < iterations; j++) {
      __asm__ __volatile__(
          "STR %1, [%2, #64]" "\n\t"
                ...
                ...
          "STR %1, [%2, #128]"  "\n\t"
          "STR %1, [%2, #2048]" "\n\t" //Total of 32 stores
    instructions
          "ADD %0, %2, #2048"   "\n\t"
          : "=r"(st_pointer)
          : "r"(useless_data), "r"(st_pointer)
          );

        if ((int)st_pointer+(32*64) > (int)st_array + (DCL2_SIZE
    *2)
          st_pointer = st_array;
        else
          st_pointer += 0;
    }

  free(st_array);
  return 0;
}
```

Algorithm 9 Microbenchmark for store misses - Cortex R5

```
int store_miss(unsigned long int size, const int iterations)
{
    size = size / sizeof(int*);
    int *st_array = (int*) malloc (size*2*sizeof(int*));
    int *st_pointer = st_array;
    int useless_data = 0xdeadbeef;

    register int j;
    for (j = 0; j < iterations; j++) {
      __asm__ __volatile__(
          "STR %1, [%2, #32]" "\n\t"
          "STR %1, [%2, #64]" "\n\t"
                ...
                ...
          "STR %1, [%2, #2048]" "\n\t" //Total of 64 stores
    instructions
          "ADD %0, %2, #2048"    "\n\t
          : "=r"(st_pointer)
          : "r"(useless_data), "r"(st_pointer)
          );

        if ((int)st_pointer+(32*64) > (int)st_array + (DC_SIZE
    *2)
          st_pointer = st_array;
        else
          st_pointer += 0;
    }

  free(st_array);
  return 0;
}
```

5.2.3 Data prefetcher

The ARM v8 and v7 architectures have implemented a Data prefetcher which changes consistently the behavior of the processor under study when the mi-

crobenchmarks are performed. In fact, it can anticipate the fetch of a cache line in the memory cache when a cache miss occurs. In general, it is hard to determine how many L1 and L2 cache and memory accesses will perform the prefetcher, thus creating both arbitrary interference on other cores, and altering in unpredictable ways the sensitivity to interference of the task under analysis. Therefore, it is needed to disable it during operation for the sake of time predictability. Thus, it must also be disabled before running the microbenchmarks described previously in this chapter. In order to disable such prefetcher in both processor cluster architectures, two functions were devised: one to be employed for the Cortex A53 processor cores (Algorithm 10) and the other one to be used for the Cortex R5 processor cores (Algorithm 11), which are written in both C/C++ and assembly programming languages.
Focusing on the first one, it can be seen that firstly the content of the *CPU Auxiliary Control Register* of 64-bits is read and stored in the `r` variable. Afterwards, a mask with all bits set except the one of the prefetcher (`mask`) is operated with a bitwise AND operation with the register in `r`, so that its contents are preserved except the prefetch bit, which is reset. As a last step, this variable overwrites the content of the register read before, thus disabling the Data prefetcher.
An analogous procedure is followed in the second algorithm to disable the data prefetcher in the Cortex R5 processor cores. In fact, in this case, the assembly code is different from the other one since that the architecture is different as well. Once the variable `leggi` contains the bit values of the ACTRL register of 32-bits (*Auxiliary Control Register*), it is operated with a bitwise OR with the corresponding mask (`Mask_R5_Dis_prefetch`) to set the bits that disable the prefetcher. Note that this last variable is a bit-mask defined as constant that changes two specific bits from zero to one in order to achieve the disabling of such data prefetcher as explained in the manual [24]. After this operation, the `leggi` variable overwrites the content of the ACTRL register, disabling the Data prefetcher.
Note that the Data prefetcher of both Cortex A53 and Cortex R5 are enabled

again at the end of the PMC function devised to read the PMCs described in section (4.2), which will be described later, for the sake of returning the platform to its initial state. In the system during operation, prefetchers would be disabled in all cores before starting the execution of critical real-time tasks in any core. Then, whenever those tasks would finish, prefetchers could be set back to maximize average performance of non-critical software. The command lines used for enabling the Data prefetcher for Cortex A53 and Cortex R5 processors are very similar to those in Algorithms 10 and 11 respectively. The only difference between the respective functions is that, in the case of the Cortex A53 cores, the `&=` logic operation is substituted with the `|=` one in the Data prefetcher enabling function. Regarding the Cortex R5 processor, instead, firstly the complement of the constant bit-mask is computed and then the `&=` logic operation is used instead of the `|=` operator.

Algorithm 10 Disabling Data Prefetcher - A53

```
#define mask_DPreFetch 0xA000 //Bit mask for Disabling data
    prefetching

void Disable_Prefetch()
{
  unsigned long long r = 0;
  unsigned long long mask = ~mask_DPreFetch;

  __asm__ __volatile__( "MRS %0, S3_1_C15_C2_0" : "=r"(r)); //
    Read EL1 CPU Auxiliary Control Register

  r &= mask;

  __asm__ __volatile__("MSR S3_1_C15_C2_0, %0" : :"r"(r));  //
    Write EL1 CPU Auxiliary Control Register
}
```

Algorithm 11 Disabling Data Prefetcher - R5

```
#define Mask_R5_Dis_prefetch 0x3000 //Bit mask for Disabling
    data prefetching

void Disable_Prefetch()
{
  unsigned int leggi = 0;

  __asm__ __volatile__("MRC p15, 0, %0, c1, c0, 1" : "=r"(leggi)
    ); // Read ACTLR

  leggi |= Mask_R5_Dis_prefetch;

  __asm__ __volatile__("MCR p15, 0, %0, c1, c0, 1" : :"r"(leggi)
    );  // Write ACTRL
}
```

5.2.4 Events counting: PMCs

To monitor cache hits or misses of the target cache, it's needed to use the Performance Monitoring Counters defined in the specific processor architecture.
The C function defined to perform events counting is shown in Algorithm 12, which has as input arguments the initialized array with Pointer Chasing technique (`p`) and the number of times that the microbenchmark has to be performed (`nruns`) that is chosen by the user.
Firstly, some arrays have to be defined in such function:

- In `PMCs[N° Events]`, each array element corresponds to one PMC and the array is initialized with the identifiers of the events to monitor.
- `PMCs_start[N° Events]` contains the value of each PMC at the time of start counting the corresponding events defined in `PMCs[N° Events]`.

- `PMCs_stop[N° Events]`, instead, contains the value of each PMC after the execution of the microbenchmark for the corresponding events defined in `PMCs[N° Events]`.

Note that `t_PMC_data` and `t_cnt_values` are defined in the overall code as `unsigned int` and `long long` type respectively.
After these first steps, the prefetcher is disabled, and invalidation of the instruction cache and flush of the data cache are performed thanks to the `Xil_ICacheInvalidate` and `Xil_DCacheFlush` functions defined in the *Xilinx* environment. Then, the PMCs are enabled to start counting and their initial values are saved in `PMCs_start[]`, and immediately after the desired microbenchmark is executed. Upon the completion of the microbenchmark, the PMCs are disabled and their values retrieved to `PMCs_stop[]`. Then, the `for` loop iterates across the different PMCs monitored printing their identifiers and the number of events occurred during the microbenchmark execution, which is equal to the difference between `PMCs_stop[j]` and `PMCs_start[j]`. Finally, flush and invalidation of the relative caches are performed again and the Data prefetcher can be brought to his initial state, namely enabled.
It's important to note that data and instructions are processed continuously in the processor and they can affect the PMCs counting, meaning that the number of events counted can vary due to effects not strictly related to the execution of the microbenchmark. Therefore, the best solution to mitigate measurement noise as much as possible is to enable PMCs right before the microbenchmark loop (inside the microbenchmark) and disable them right after. For the sake of illustration, however, we indicate PMC enabling and disabling outside the microbenchmark, as this facilitates understanding the operation of the whole process. In practice, those operations occur inside the microbenchmarks themselves.

Algorithm 12 Performance Monitoring Counter function

```
int PMC(register int** p, int nruns)
{
  register int i, j, status;

  t_PMC_data PMCs[N° Events] = {A,B,...};
  t_cnt_values PMCs_start[N° Events];
  t_cnt_values PMCs_stop[N° Events];
  uint32_t len = N° Events;

  Disable_Prefetch();

  Xil_ICacheInvalidate();
  Xil_DCacheFlush();

  start_PMCs(PMCs_start[]);

  status = Microbenchmark(p, nruns); // Target microbenchmark to
     be monitored

  stop_PMCs(PMCs_stop[]);

  for (j=0; j<len; j++){ // Reading PMCs
    printf ("#PMCs: %x\n", PMCs[j]);
    printf (" %lld \t # low\n", PMCs_stop[j] - PMCs_start[j]);
  }

  Xil_ICacheInvalidate();
  Xil_DCacheFlush();

  Enable_Prefetch();

  return status;
}
```

Chapter 6

Results

In this chapter, the experimental results obtained with the microbenchmarks described in section 5.2 are shown. Firstly, results for the execution on isolation are reported. Then, results with contenders are discussed, along with the research findings.

6.1 Experiments in Isolation

Several experiments of tasks in isolation have been performed for this Master thesis. Given that four Cortex-A53 cores are included in the APU of the Zynq platform and two Cortex-R5 cores are included instead in the RPU, having A53 and R5 cores and clusters different characteristics, we need to assess their performance and cache features separately. Therefore, in the following, experimental results for one core in each of the clusters, both of the APU and RPU, will be represented and discussed. For the sake of convenience, we refer to the core analyzed as the "main core", although all cores in each cluster are identical.

6.1.1 Cortex-A53 laboratory results

In the following, the results obtained for the main core of the Cortex-A53 processor are analyzed in detail, since it is mandatory to assess the reliability of the microbenchmarks implemented, their performance on just one core and whether there are glitches on the board under study (i.e. whether actual behavior matches specifications).

Level-1 data cache: accessing different sets

The first microbenchmark performed in this work is the one aimed to access different sets of the target cache (Algorithm 4). Then, changing the array size while keeping the stride value constant, read cache hits and misses can be experienced, as explained in previous chapter.

L1 read cache hits. To obtain L1 data cache hits accessing different sets, fixed array size of 24 kB and stride of 64 bytes were set, getting the results shown in table 6.1.
The number obtained in the laboratory of the L1D_CACHE event, which is one of the events of the *PMC events* discussed in section 4.2, is higher than the one expected. In fact, before the Load instructions are performed, accesses to L1 data cache occur due to other instructions needed for the implementation of the microbenchmark, which justifies this small discrepancy.
Looking at the APU_L2D_CACHE event, the expected results match quite well the ones obtained in laboratory since that no access to Level-2 cache is required and it is just the sum of two PMCs already addressed, namely L1D_CACHE_REFILL and L1I_CACHE_REFILL.
In fact, the first one is equal to 384, which is expected since that:

$$\frac{Array\ size}{stride} = \frac{24 \cdot 1024\ Bytes}{64\ Bytes} = 384 \tag{6.1}$$

Note that L1I_CACHE_REFILL is always equal to 11 for all the iterations

performed. Moreover, in this case, the APU_L2D_CACHE_REFILL event matches the previous one named APU_L2D_CACHE, as expected, since no dirty data is evicted from any cache memory.
For what concerns the remaining events named APU_EXT_MEM_REQUESTS and APU_MEM_ACCESS, they also match the expected results. In fact, the first one gives a result that is the sum of the L1D_CACHE_REFILL and L1I_CACHE_REFILL events, which is expected since that external memory requests cause cache line refills of both L1 data and instruction cache. The second one, instead, is counting not only the cache line refills of the first level of cache, but also other instructions that are needed for the implementation and execution of the microbenchmark employed.
When more iterations are performed of the same microbenchmark, it's possible to note that the results obtained for some PMCs are almost equal to the ones obtained with just one iteration. For such counters, which are related to data cache accesses and cache line refills of the two cache levels implemented in Cortex-A53, is expected that the number of events counted doesn't change. In fact, the size of the initialized array completely fits in the Level-1 data cache and no other cache line refills or external memory requests are performed.
Counting of memory and L1 data cache accesses, instead, increase as expected, since that with higher number of iterations, more accesses to the first level of cache have to performed.
In terms of CPU_CYCLES, we observe that they are around 50,000 for just one iteration, thus more than 100 cycles per memory instruction (L1D_CACHE). This is expected because with just one iteration all accesses miss in cache. When increasing iterations to 10, then execution time is around 50,000 cycles plus 3 cycles per hit. Since there are in the order of 3,500 cache hits (3,500 additional L1D_CACHE accesses), we can conclude that each L1 data cache hit costs around 3 cycles to lead to the 10,000 cycles increase w.r.t. 1 iteration. As we increase the number of iterations,

the same conclusion holds, being execution time (in cycles) around 3 times L1D_CACHE plus 50,000 cycles.
Overall, we can conclude that, as expected, this microbenchmarks causes all-misses during the first iteration and all-hits during the following ones.

L1 (read) Hits - A53_0			
Iterations	PMC events	Lab. results	Expected results
1	L1D_CACHE	397	384
	APU_L2D_CACHE	395	384+11
	L1D_CACHE_REFILL	384	384
	APU_L2D_CACHE_REFILL	395	384+11
	CPU_CYCLES	50728	-
	APU_MEM_ACCESS	408	>384
	APU_EXT_MEM_REQUESTS	394	384+11
10	L1D_CACHE	3853	3840
	APU_L2D_CACHE	394	384+10
	L1D_CACHE_REFILL	384	384
	APU_L2D_CACHE_REFILL	394	384+10
	CPU_CYCLES	60703	-
	APU_MEM_ACCESS	3864	>3840
	APU_EXT_MEM_REQUESTS	394	384+10
100	L1D_CACHE	38413	38400
	APU_L2D_CACHE	394	384+10
	L1D_CACHE_REFILL	384	384
	APU_L2D_CACHE_REFILL	394	384+10
	CPU_CYCLES	164439	-
	APU_MEM_ACCESS	38424	>38400
	APU_EXT_MEM_REQUESTS	394	384+10
1000	L1D_CACHE	384013	384000
	APU_L2D_CACHE	394	384+10
	L1D_CACHE_REFILL	384	384
	APU_L2D_CACHE_REFILL	394	384+10
	CPU_CYCLES	1202360	-
	APU_MEM_ACCESS	384024	>384000
	APU_EXT_MEM_REQUESTS	394	384+10

Table 6.1: L1 reads cache hits accessing different sets - A53_0

L1 read cache misses. For what concerns L1 read cache misses, it's needed to employ an array size bigger than the size of such data cache. Since that L1 data cache of Cortex-A53 is equal to 32 kB, an array size of 40 kB is a reasonable choice. Therefore, selecting this array size with 64 bytes of stride, results in table 6.2 are obtained.

In the first iteration, similar results are obtained for events like L1D_CACHE and APU_MEM_ACCESS compared with the ones represented in table 6.1. The only relevant difference is that, since a larger array is traversed (40 kB instead of 24 kB), the number of accesses, and so the values of these event counters, increase proportionally. In any case, this result is fully expected since the first iteration performs all-misses in both cases.

It is also noted that the APU_L1D_CACHE_WB counter is not zero. This is an unexpected result since array data is only read and never modified, so cache lines evicted from the L1 data cache are clean. Thus, we would expect this counter to be zero, but it is not. In fact, it corresponds exactly to the number of array elements that don't fit in the L1 data cache. The total number of array elements is 640 using the expression 6.1, but the L1 data cache is 32 kB, meaning that just 512 array elements can be stored without evictions. Therefore, making the difference between these two last numbers, 128 is the number of array elements that are evicted from L1 to higher memory levels with 1 iteration, and the value read for this counter is 134, so with only a negligible discrepancy due to other instructions in the microbenchmark. As we increase the number of iterations, it holds that APU_L1D_CACHE_WB is roughly equal to L1D_CACHE_REFILL minus 512. Overall, a relevant conclusion of this analysis is as follows: *APU_L1D_CACHE_WB does not count only dirty L1 data evictions, but* ***all*** *evictions (dirty and clean), which does not match the specifications.*

Note that the counted number of L2 data cache accesses is quite consistent and this is due to the fact that such counter includes both L1 data misses (L1D_CACHE_REFILL) and L1 data cache write-backs (APU_L1D_CACHE_WB), as explained in section 4.2. Moreover, the num-

ber of events counted for L1I_CACHE_REFILL PMC is always equal to 10 for each number of iterations performed.
Note that the number of L1 data cache misses (L1D_CACHE_REFILL) does not match the number of L1 data cache accesses (L1D_CACHE). In fact, the target number of misses for this microbenchmark is as shown in the following expression:

$$N^\circ\ misses_{total\ iterations} = N^\circ\ misses_{1\ iteration} \cdot N^\circ\ iterations \tag{6.2}$$

Such relation has been proven to hold in other processors where the cache replacement policy is LRU. However, since the L1 data cache replacement policy implemented is Pseudo-Random in our case, many cache hits are experienced. In our microbenchmark, by traversing an array of 40 kB, we place 5 different cache lines (namely A, B, C, D and E) in each 4-way set. Under LRU, we would fetch A, B, C, D first, then E would evict A, A would evict B, B would evict C and so on an so forth so that each access would evict the cache line needed next, thus creating an all-misses patterns. In the case of pseudo-random replacement, once A, B, C, D have been fetched, E may or may not evict A, so that some times it leads to a miss and some others to a hit. For instance, if E evicts C, A will hit, B will hit and C will miss, thus leading to another eviction, which may or may not evict D. Overall, a relevant fraction of hits is expected, which matches with the results obtained. In particular, as we increase the number of iterations, the miss rate approaches 40% instead of the 100% desired.
Therefore, a different experiment is needed to cause an all-misses behavior, which allows us to maximize contention in the desired shared hardware resources.

L1 (read) Misses - A53_0			
Iterations	PMC events	Lab. results	Expected results
1	L1D_CACHE	653	640
	APU_L2D_CACHE	784	640+10+134
	L1D_CACHE_REFILL	640	640
	APU_L2D_CACHE_REFILL	650	640+10
	CPU_CYCLES	83902	-
	APU_MEM_ACCESS	664	>640
	APU_L1D_CACHE_WB	134	134
10	L1D_CACHE	6413	6400
	APU_L2D_CACHE	5440	>2968+10+2461
	L1D_CACHE_REFILL	2968	6400
	APU_L2D_CACHE_REFILL	650	640+10
	CPU_CYCLES	130327	-
	APU_MEM_ACCESS	6424	>6400
	APU_L1D_CACHE_WB	2461	5894
100	L1D_CACHE	64013	64000
	APU_L2D_CACHE	51471	>25984+10+25477
	L1D_CACHE_REFILL	25984	64000
	APU_L2D_CACHE_REFILL	651	640+10
	CPU_CYCLES	602222	-
	APU_MEM_ACCESS	64024	>64000
	APU_L1D_CACHE_WB	25477	63494
1000	L1D_CACHE	640013	640000
	APU_L2D_CACHE	512463	>256480+10+255973
	L1D_CACHE_REFILL	256480	640000
	APU_L2D_CACHE_REFILL	650	640+10
	CPU_CYCLES	5335666	-
	APU_MEM_ACCESS	640024	>640000
	APU_L1D_CACHE_WB	255973	639494

Table 6.2: L1 reads cache misses accessing different sets - A53_0

Level-1 data cache: accessing the same set

The microbenchmark devised to cause all-misses in the L1 data cache with the implemented Pseudo-Random replacement policy is the one aimed to force many accesses to occur on the same set of cache lines (Algorithm 5). Such algorithm forces all cache lines accessed be placed in the same set so that few different cache lines are enough to exceed the space in a cache set. While we cannot enforce all accesses to be misses due to the Pseudo-Random replacement policy, where the probability of survival of a cache line is never zero, we can approach asymptotically such case by increasing the number of cache lines largely above the cache set space.
Therefore, employing the One-Way stride value (5.1) and adjusting the array size with respect to the number of replacements to be performed, the results in table (6.3) were obtained. In such experiment, "replacements" stands for the number of cache lines fetched in excess of the cache space. For instance, 4 replacements means that we access 8 different cache lines for a 4-way cache, thus making sure that at least 4 replacements occur.
As shown the table, the laboratory results of L1D_CACHE matches always the expected ones since that each load instruction has to cause one access to the L1 data cache. Note that the laboratory results of such counter are slightly bigger than the expected ones (by 14 events) because of other L1 data cache accesses performed before the load instructions implemented in the specific microbenchmark.
From the laboratory results of the L1 cache line refills counter, instead, it's possible to figure out that the cache miss rate increases with higher number of replacements performed in the same set of cache lines. In fact, increasing the number of replacements, the laboratory results are getting closer to the expected ones, meaning that the probability of cache misses are increasing as well while cache hits are less likely. In the most extreme case, with 116 replacements the miss rate is around 99%, since 120 different cache lines contend for 4 physical cache lines in the set.

L1 Misses: accessing same set - A53_0			
Replacements	PMC events	Lab. results	Expected results
4	L1D_CACHE	8014	8000+14
	L1D_CACHE_REFILL	6122	8000
	CPU_CYCLES	222453	-
12	L1D_CACHE	16014	16000+14
	L1D_CACHE_REFILL	14501	16000
	CPU_CYCLES	538315	-
20	L1D_CACHE	24014	24000+14
	L1D_CACHE_REFILL	22502	24000
	CPU_CYCLES	2172028	-
28	L1D_CACHE	32014	32000+14
	L1D_CACHE_REFILL	30545	32000
	CPU_CYCLES	4567597	-
36	L1D_CACHE	40014	40000+14
	L1D_CACHE_REFILL	38546	40000
	CPU_CYCLES	6560729	-
76	L1D_CACHE	80014	80000+14
	L1D_CACHE_REFILL	78547	80000
	CPU_CYCLES	13714240	-
116	L1D_CACHE	120014	120000+14
	L1D_CACHE_REFILL	118546	120000
	CPU_CYCLES	20732514	-

Table 6.3: L1 reads cache misses accessing the same set - A53_0

Level-1 data cache: store instructions

Another type of microbenchmark was devised to perform writes to the L1 data cache as described in detail in section (5.2.2). Note that two different microbenchmarks have to be run because of the different stride values that it's needed to cause hits or misses in the cache.

L1 write cache hits. Using the microbenchmark to perform memory write operations aimed to cause cache hits (Algorithm 6), the results shown in table (6.4) were obtained. Note that for each case of the number of iterations defined, the stride value from each store instruction was set to 64 bytes and a total of 32 store instructions are performed in each iteration.
Firstly, with just one iteration, the L1D_CACHE counter generated a number that is high compared with to the expected value (32 store instructions). In fact, it was experimentally tested that there are 20 accesses to L1 data cache without considering the 32 store instructions of the microbenchmark implemented, being 6 of them within the main loop. Then, it's possible to conclude that the laboratory results match the expected ones.
For what concerns the counter related to the accesses to the L2 data cache, it can be observed that it includes also the sum of the instruction and data cache line refills counters represented in the same table. In fact, the L1I_CACHE_REFILL counter is always equal to 7 for each case of iterations. Similar observations can be made regarding the L2 data cache line refills and the external memory requests counters. The first one matches the expected results, since that it is the sum of the data and instruction L1 data cache line refills counters, while the second one matches, as expected, the number of accesses performed to the second level cache, which is external to the inner one.
Note that the number of cache misses (L1D_CACHE_REFILL) is 32 for 1 iteration, thus reflecting the 32 misses due to the 32 store instructions, which fetch 2 kB of data. When we increase iterations to 10, there are 320 misses (20 kB of data). For 100 iterations we have 383 misses ($\approx$24 kB of data). In this latter case, we fetch the whole array several times, so we experience the maximum number of misses possible. Finally, for 1,000 iterations the number of misses remains constant (383), so the increase on execution time w.r.t. the case with 100 iterations is caused only due to store hits in the L1 data cache. Therefore, there is an increase in execution time of around 55,000 cycles caused by 900 iterations with 32 store operations each, which

leads to an overall latency of 2 cycles to process each store hit operation on average.

Note also that, by having 6 "unwanted" memory accesses within the loop, the L1D_CACHE counter matches approximately 32+6 accesses per iteration. Those 6 accesses correspond to the code checking the condition to update the array pointer and few other variable accesses. They could likely become register accesses with some compiler optimizations. However, compilation was performed without optimizations to prevent the compiler from altering the assembly code placed to create specific cache behavior.

L1 (write) Hits - A53_0			
Iterations	PMC events	Lab. results	Expected results
1	L1D_CACHE	52	32+6+14
	APU_L2D_CACHE	38	32+6
	L1D_CACHE_REFILL	32	32
	L1I_CACHE_REFILL	7	∼7
	APU_L2D_CACHE_REFILL	37	∼38
	CPU_CYCLES	1172	-
	APU_EXT_MEM_REQUESTS	38	∼37
10	L1D_CACHE	394	320+60+14
	APU_L2D_CACHE	390	320+6
	L1I_CACHE_REFILL	7	∼7
	L1D_CACHE_REFILL	320	320
	APU_L2D_CACHE_REFILL	326	∼326
	CPU_CYCLES	9672	-
	APU_EXT_MEM_REQUESTS	326	∼326
100	L1D_CACHE	3830	3200+600+14
	APU_L2D_CACHE	390	383+7
	L1I_CACHE_REFILL	7	∼7
	L1D_CACHE_REFILL	383	384
	APU_L2D_CACHE_REFILL	388	∼390
	CPU_CYCLES	16828	-
	APU_EXT_MEM_REQUESTS	326	∼326
1000	L1D_CACHE	38180	32000+6000+14
	APU_L2D_CACHE	390	383+7
	L1I_CACHE_REFILL	7	∼7
	L1D_CACHE_REFILL	383	384
	APU_L2D_CACHE_REFILL	388	∼390
	CPU_CYCLES	71671	-
	APU_EXT_MEM_REQUESTS	389	∼390

Table 6.4: L1 writes cache hits - A53_0

L1 write cache misses. To cause L1 cache misses on memory write operations, the microbenchmark based on store instructions with a stride value of 64 bytes is needed, namely one cache line length, which is explained in detail in section (5.2.2). Using a total of 32 store instructions for just one iteration, the results shown in table (6.5) are obtained.
Considering the case when the microbenchmark is executed just once (iteration), the laboratory results of the L1 data cache access counter are analogous to those for the L1 write cache hits microbenchmark, since both microbenchmarks cause all-misses behavior for just one iteration. Similar conclusions can be reached for 10 iterations since both microbenchmarks still cause all-misses behavior.
When increasing iterations to 100, instead, the number of misses in the L1 data cache (L1D_CACHE_REFILL counter) increases to 3,200, thus highly in line with the 32 store misses per iteration of the microbenchmark. We also note that the number of L1 data cache accesses (L1D_CACHE counter) includes 6 hits per iteration as in the all-hits case.
Regarding L2 accesses, APU_L2D_CACHE, we observe that it includes two types of accesses: L1 cache misses (mostly L1D_CACHE_REFILL counter) and L1 data cache write-backs (APU_L1D_CACHE_WB counter), being the latter indicated as `wb` in the table. The former corresponds to data requested from L1 caches, whereas the second corresponds to data evicted from the L1 data cache. In the case of 1 and 10 iterations, the amount of data accessed is 2 kB and 20 kB respectively, thus not enough to fill the cache (32 kB), so not causing any L1 data cache eviction. However, after 16 iterations the L1 data cache gets full, as shown with the expression below:

$$Data\_size = N°iterations \cdot N° \frac{stores}{iteration} \cdot stride = 16 \cdot 32 \cdot 64 = 32kB \quad (6.3)$$

Therefore, except those first 512 stores (32 stores per iteration during 16 iterations), all remaining stores cause an L1 data cache eviction, and so an additional L2 cache access. Hence, in the case of 100 iterations we would ex-

pect 3,200 L2 accesses due to store misses and 3,200-512=2,688 L2 accesses due to L1 data cache line eviction. Thus, the total number of observed L2 accesses (5,910) matches quite well the expectations (3,200+2,688=5,888). Results for 1,000 iterations follow the same trends with 32,000 L1 data cache misses, 31,500 L1 data cache write-backs and 63,500 L2 cache accesses.

A somewhat unexpected result corresponds to the APU_L2D_CACHE_WB counter (not shown in the table), which is always 0. The L2 cache has 1 MB capacity. Hence, when using the microbenchmark with 1, 10 and 100 iterations we are unable to fill it and therefore, no evictions occur. However, when performing 1,000 iterations, 2 MB of data are fetched, thus meaning that the latest half of the 32,000 store accesses should produce L2 evictions. If this was the case, APU_L2D_CACHE_WB would be 16,000 and APU_EXT_MEM_REQUESTS would be 48,000 (32,000 misses + 16,000 evictions). However, L2 evictions are neither counted by the L2 evictions counter nor by the memory access counter. This indicates that either those events are not monitored correctly or their configuration (as described in the specifications) is not properly described. This has been double-checked and no error was found in the code, which configures the corresponding PMCs as indicated in the specification. Also, the fact that this issue affects two different event counters provides some indication that the event may not be monitored properly by the hardware, thus not counting it as expected. Overall, another relevant conclusion is that *APU_L2D_CACHE_WB and APU_EXT_MEM_REQUESTS event counters may not work properly.*

Finally, we want to note that the execution time (in cycles of this microbenchmark reaches 1,220,000 cycles approximately for 32,000 store misses. This leads to the conclusion that the processor can perform a store operation in memory every 40 cycles on average. This is in contrast with load latencies, which were largely above 100 cycles, thus reflecting that stores can be processed offline to some extent without stalling execution despite the fact that memory latency may be above 100 cycles.

L1 (write) Misses - A53_0			
Iterations	PMC events	Lab. results	Expected results
1	L1D_CACHE	48	32+6+10
	APU_L2D_CACHE	38	32+6+wb
	L1D_CACHE_REFILL	32	32
	APU_L2D_CACHE_REFILL	37	31+6
	CPU_CYCLES	1191	-
	APU_EXT_MEM_REQUESTS	38	∼38
	APU_L1D_CACHE_WB	0	0
10	L1D_CACHE	390	320+60+10
	APU_L2D_CACHE	326	320+6+wb
	L1D_CACHE_REFILL	320	320
	APU_L2D_CACHE_REFILL	326	320+6
	CPU_CYCLES	9700	-
	APU_EXT_MEM_REQUESTS	326	∼326
	APU_L1D_CACHE_WB	0	0
100	L1D_CACHE	3810	3200+600+10
	APU_L2D_CACHE	5910	∼3200+6+wb
	L1D_CACHE_REFILL	3201	3200
	APU_L2D_CACHE_REFILL	3205	3200+6
	CPU_CYCLES	93470	-
	APU_EXT_MEM_REQUESTS	3207	∼3206
	APU_L1D_CACHE_WB	2703	∼2688
1000	L1D_CACHE	38010	32000+6000+10
	APU_L2D_CACHE	63508	∼32000+6+wb
	L1D_CACHE_REFILL	32000	32000
	APU_L2D_CACHE_REFILL	32005	∼32000
	CPU_CYCLES	1220134	-
	APU_EXT_MEM_REQUESTS	32006	∼32006
	APU_L1D_CACHE_WB	31503	∼31488

Table 6.5: L1 writes cache misses - A53_0

6.1.2 Cortex-R5 laboratory results

The experimental results obtained for the main core of the RPU that includes the two Cortex-R5 cores are presented next.

Level-1 data cache: accessing different sets

Focusing on the memory read operations, firstly accesses to different sets were performed through the use of the same microbenchmark used for Cortex-A53 (Algorithm 4). Therefore, the same reasoning regarding the array size and the stride value for Cortex-A53 holds also for the ARM v7 architecture, with the only difference that cache line size differs across Cortex A53 and R5 cores, as explained before.

L1 read cache hits. L1 read data cache hits are obtained using a fixed array size of 24 kB and stride of 32 bytes, which corresponds to one cache line length, obtaining the results shown in table (6.6).
As it can be seen, the number of L1 data cache accesses matches almost perfectly the expected one. The only difference between the practical and the theoretical results of the L1D_CACHE counter is that there are other accesses to the L1 data cache that are related just to the implementation of the microbenchmark.
Considering the formula about computing the number of cache line refills (Equation 6.1), it can be highlighted that the results obtained for the counter of the cache line refills experienced by L1 data cache matches the expected ones, since that 768 is the number of load instructions performed by the specific microbenchmark: 768 loads with stride 32, for a total of 24 kB. Therefore, it corresponds also to the number of cache line refills (misses) that L1 data cache is experiencing.
Then, the number of external memory requests matches the one of the cache line refills since, in the first iteration, the processor has to fetch all the array elements from the main memory, thus being all accesses misses.
Afterwards, as we increase the number of iterations, since that the size of the

array completely fits in the L1 data cache, the number of L1 data cache line refills and external memory requests doesn't change. As expected, instead, more accesses to the L1 data cache are performed increasing the number of iterations and the numbers obtained match always the expected results.

L1 (read) Hits R5_0			
Iterations	PMC events	Lab. results	Expected results
1	L1D_CACHE	778	768+10
	L1D_CACHE_REFILL	768	768
	L1I_CACHE_REFILL	18	∼18
	CPU_CYCLES	43193	-
	RPU_EXT_MEM_REQUESTS	768	768
10	L1D_CACHE	7690	7680+10
	L1D_CACHE_REFILL	768	768
	L1I_CACHE_REFILL	18	∼18
	CPU_CYCLES	64318	-
	RPU_EXT_MEM_REQUESTS	768	768
100	L1D_CACHE	76810	76800+10
	L1D_CACHE_REFILL	768	768
	L1I_CACHE_REFILL	18	∼18
	CPU_CYCLES	275546	-
	RPU_EXT_MEM_REQUESTS	768	768
1000	L1D_CACHE	768010	768000+10
	L1D_CACHE_REFILL	768	768
	L1I_CACHE_REFILL	18	∼18
	CPU_CYCLES	2387851	-
	RPU_EXT_MEM_REQUESTS	768	768

Table 6.6: L1 reads cache hits accessing different sets - R5_0

L1 read cache misses. The array size chosen to cause cache misses in Level-1 cache of the main core Cortex-R5 is 40 kB with stride value of 32 bytes, obtaining the results shown in table 6.7, noting that Pointer Chasing

technique is employed.
When just one iteration is performed, the number of accesses to the L1 data cache counted by L1D_CACHE is almost equal to the one predicted theoretically. In fact, in this case, recalling Equation 6.1, 1280 is the number of array elements and it corresponds to the number of cache lines that L1 data cache should handle. The remaining 10 accesses correspond to other memory instructions in the microbenchmark excluding the loads place on purpose in the loop.
When considering the data cache line refills and the external memory requests counted with 1 iteration, it can be noted that the results match the expected ones: around 1,280 L1 data cache misses and memory accesses to fetch 40 kB of data.
As we increase the number of iterations, we observe some trends analogous to those of the Cortex-A53 core case: a large number of loads hits in the L1 data cache due to the pseudo-random replacement policy, with the miss rate being around 40%. For instance, with 10 iterations we have around 12,800 L1 data cache accesses (L1D_CACHE), out of which 5,380 approximately are L1 data cache refills (L1D_CACHE_REFILL) and main memory accesses (RPU_EXT_MEM_REQUESTS).
Another relevant information is that the number of L1 data cache write-backs is virtually zero. This is in contrast with the case of the Cortex-A53 core, where the number of write-backs was similar to the number of refills despite cache lines being clean. We regard the case of the Cortex-R5 core as the correct case, since it is the one matching the description in the specifications.

L1 (read) Misses R5_0			
Iterations	PMC events	Lab. results	Expected results
1	L1D_CACHE	1290	1280+10
	L1D_CACHE_REFILL	1281	1280
	L1I_CACHE_REFILL	17	∼18
	CPU_CYCLES	72057	-
	RPU_EXT_MEM_REQUESTS	1286	1280
	RPU_DCACHE_WB	3	0
10	L1D_CACHE	12810	12800+10
	L1D_CACHE_REFILL	5382	12800
	L1I_CACHE_REFILL	17	∼18
	CPU_CYCLES	324630	-
	RPU_EXT_MEM_REQUESTS	5388	12800
	RPU_DCACHE_WB	6	0
100	L1D_CACHE	128010	128000+10
	L1D_CACHE_REFILL	51462	128000
	L1I_CACHE_REFILL	18	∼18
	CPU_CYCLES	3118184	-
	RPU_EXT_MEM_REQUESTS	51721	128000
	RPU_DCACHE_WB	6	0
1000	L1D_CACHE	1280010	1280000+10
	L1D_CACHE_REFILL	512262	1280000
	L1I_CACHE_REFILL	18	∼18
	CPU_CYCLES	31055075	-
	RPU_EXT_MEM_REQUESTS	512534	1280000
	RPU_DCACHE_WB	5	0

Table 6.7: L1 reads cache misses accessing different sets - R5_0

Level-1 data cache: accessing the same set

In the case of R5 processor, the same microbenchmark (5) used for the Cortex A53 core was employed in order to access the same set of cache lines

in the L1 data cache. Therefore, maintaining the same stride value of 8 kB between each replacement, namely the distance of one way w.r.t. the next one, and changing the array size depending on the number of replacements to be implemented, it was possible to get the results shown in table 6.8.
The laboratory results obtained for the L1D_CACHE matches always the expected ones, with the minor difference caused by other (few) memory accesses excluding the load pattern tested.
Regarding the counter of cache line refills (misses), it is surprisingly always equal to the number of times that the array elements are read from the main memory. This is an unexpected behavior of the L1 data cache of R5 since, given that it implements a pseudo-random replacement policy, we would expect results analogous to those of the Cortex-A53 core, where an increasing number of replacements leads to a miss rate approaching asymptotically 100%. However, as shown in these experiments, with 4 replacements (8 lines competing for 4 ways in the set), all accesses become misses. While this is counterintuitive, it is possible and compatible with the fact that our results for the read misses case reveal that the replacement policy is pseudo-random. In particular, in the previous experiment we traversed a 40 kB array, thus placing 5 cache lines per set (thus with 1 replacement). Depending on how the pseudo-random replacement policy is implemented, it may lead to systematic scenarios where 1 replacement is random, but 4 replacement systematically evict all 4 lines in the set.
While the actual implementation is unknown, a recent work has proposed a replacement policy with exactly those characteristics [25]. In particular, such policy proposes to perform a random permutation of the cache ways to select the next cache line to be evicted. Hence, the first choice is random, the second choice is random among the non-evicted lines, and so on and so forth until the N^{th} replacement (where N stands for the number of cache ways) evicts systematically the only way not evicted so far, and a new random permutation is generated. Such a policy, therefore, would evict 1 random line with 1 replacement, and all 4 lines in the set with 4 replacements (in random

order).

Overall, while using different implementations for the same replacement policy across core types is somewhat unexpected, it is possible in general, and thus we accept this hypothesis as the most likely one, and the one compatible with our measurements. Hence, Cortex-A53 and R5 cores differ on the behavior of some PMCs and on the implementation of the pseudo-random replacement policy.

L1 Misses: accessing same set - R5_0			
Replacements	PMC events	Lab. results	Expected results
4	L1D_CACHE	8011	8000+11
	L1D_CACHE_REFILL	8000	8000
	CPU_CYCLES	483899	-
12	L1D_CACHE	16011	16000+11
	L1D_CACHE_REFILL	16000	16000
	CPU_CYCLES	983038	-
20	L1D_CACHE	24011	24000+11
	L1D_CACHE_REFILL	24000	24000
	CPU_CYCLES	1680904	-
28	L1D_CACHE	32011	32000+11
	L1D_CACHE_REFILL	32000	32000
	CPU_CYCLES	2336809	-
36	L1D_CACHE	40011	40000+11
	L1D_CACHE_REFILL	40000	40000
	CPU_CYCLES	2921230	-
76	L1D_CACHE	80011	80000+11
	L1D_CACHE_REFILL	80000	80000
	CPU_CYCLES	5842789	-
116	L1D_CACHE	120011	120000+11
	L1D_CACHE_REFILL	120000	120000
	CPU_CYCLES	8764792	-

Table 6.8: L1 reads cache misses accessing the same set - R5_0

Level-1 data cache: store instructions

Analogous microbenchmarks to those devised to perform writes to the L1 data cache of Cortex A53 are employed for Cortex R5 processor, with the aforementioned difference of the stride (32 instead of 64 bytes) and the number of stores in the loop (64 instead of 32). For this reason, two different experiments have to be distinguished, which are the one that causes cache store hits and the one that causes cache store misses in the L1 data cache.

L1 write cache hits. The results shown in table (6.9) were obtained employing the microbenchmark aimed to cause cache hits with store instructions (Algorithm 6). It's important to observe that, in this case, the stride value from each store instruction was set to 32 bytes and a total of 64 store instructions are performed in each iteration as explained before.
Considering the case with just one iteration, the L1D_CACHE PMC increases more than expected and this is due to the fact that there are 20 accesses to L1 data cache that are counted without considering the 64 store instructions of the microbenchmark implemented. Out of those 20 accesses, 6 of them are within the loop, as in the case of the Cortex-A53 cores. Then, it can be concluded that the Laboratory results match the expected ones.
The L1D_CACHE_REFILL event (L1 data cache misses) is almost equal to the expected results. Analogous conclusions can be reached for the number of memory accesses (RPU_EXT_MEM_REQUESTS). The fact those counters are slightly lower than expected is very likely because the microbenchmark finalizes its execution while some stores are still in flight and thus, not counted yet, since stores are typically processed asynchronously.
In any case, for 1 iteration the number of misses matches the expectations: 64 store misses to fetch 2 kB of data. For 10 iterations, the behavior is still all-misses since 20 kB of data are fetched and never accessed again. However, as for the Cortex-A53 core case, the full array (24 kB) is fetched after 12 iterations, thus with 768 L1 data cache misses. Hence, the remaining iterations in the cases with 100 and 1,000 iterations perform all-hits accesses.

The relation between number of accesses, the number of stores per iteration and the stride, and the data size fetched, is shown in the following equation:

$$Data\_size = N°iterations \cdot N° \frac{stores}{iteration} \cdot stride = 12 \cdot 64 \cdot 32 = 24kB \quad (6.4)$$

Overall, the first 12 iterations have an all-misses behavior and the remaining ones an all-hits behavior. Hence, by observing the increase in execution time from 100 to 1,000 iterations (258,000 cycles), we can conclude that one store hit is processed every 4.5 cycles (258,000 / (900 x 64)). Note that in the case of loads, the average hit latency was around 3 cycles for the R5 cores.

L1 (write) Hits - R5_0			
Iterations	PMC events	Lab. results	Expected results
1	L1D_CACHE	81	64+6+14
	L1D_CACHE_REFILL	61	64
	L1I_CACHE_REFILL	9	∼10
	CPU_CYCLES	3474	-
	RPU_EXT_MEM_REQUESTS	60	∼64
	RPU_DCACHE_WBACK	0	0
10	L1D_CACHE	711	640+60+14
	L1I_CACHE_REFILL	9	∼10
	L1D_CACHE_REFILL	637	640
	CPU_CYCLES	34846	-
	RPU_EXT_MEM_REQUESTS	636	∼640
	RPU_DCACHE_WBACK	0	0
100	L1D_CACHE	7019	6400+600+14
	L1I_CACHE_REFILL	9	∼10
	L1D_CACHE_REFILL	768	768
	CPU_CYCLES	66968	-
	RPU_EXT_MEM_REQUESTS	768	∼768
	RPU_DCACHE_WBACK	0	0
1000	L1D_CACHE	70094	64000+6000+14
	L1I_CACHE_REFILL	9	∼10
	L1D_CACHE_REFILL	768	768
	CPU_CYCLES	324959	-
	RPU_EXT_MEM_REQUESTS	768	∼768
	RPU_DCACHE_WBACK	0	0

Table 6.9: L1 writes cache hits - R5_0

L1 write cache misses. Results for the all-misses store microbenchmark are shown in table (6.10).

Considering the case when just 1 or 10 iterations of the microbenchmark are executed, the L1D_CACHE PMC shows the same results obtained in table (6.9), which corresponds to the microbenchmark for store hits, since the first 12 iterations experience only misses in both cases.

Using an array of 2 MB, as for the Cortex-A53 core case, would lead to no data reuse at all and hence, we would obtain all-misses behavior for 100 and 1,000 iterations. Since previous experiments revealed an abnormal behavior for the pseudo-random replacement policy in the L1 data cache, an array of 64 kB instead has been used. Such array has a size twice as large as the L1 data cache so, to some extent, has the same behavior as the 4-replacements case when accessing the same set.

The results obtained are on the one hand consistent and on the other unexpected. They are consistent since that the number of write-back operations (RPU_DCACHE_WBACK counter) plus the number of L1 data cache refills (L1D_CACHE_REFILL counter) match precisely the number of memory requests (RPU_EXT_MEM_REQUESTS counter). However, results are unexpected because the miss rate is not 100% as for the 4-replacements case when accessing a single set. In particular, miss rates for the stores are around 84% for 100 iterations (5,400 misses out of 6,400 stores), and around 75% for 1,000 iterations (48,000 misses out of 64,000 stores). This reveals that the behavior of the pseudo-random replacement policy may vary noticeably even when the number of cache lines fetched (and the pattern to access them) remains the same for any given set, as it is the case for our experiments accessing a single set or all of them.

Again, while such behavior is possible, it reveals that the pseudo-random replacement policy is far from being sufficiently random, thus leading to systematic behavior in specific scenarios. This plays against time predictability in general and hence, suggests that one cannot rely much on cache hits in the L1 data cache of this processor unless all data fits so that an all-hits behavior

is expected.

Finally, in terms of memory latency, it can be noted that one store operation in served by memory every 56 cycles on average. If we take into account that a number of store hits are served in between store misses, then this latency may be even lower. Furthermore, if we consider that each memory store operation carries out also another access due to the dirty eviction, then a memory request may be served every 28 cycles on average. This is in contrast with the case of the load misses where, for instance, 120,000 misses are served in around 8,760,000 cycles, thus averaging around 70 cycles per load miss. Hence, as in the Cortex-A53 case, stores can be served at a higher rate than loads.

L1 (write) Misses - R5_0			
Iterations	PMC events	Lab. results	Expected results
1	L1D_CACHE	81	64+6+14
	L1D_CACHE_REFILL	61	64
	L1I_CACHE_REFILL	10	∼10
	CPU_CYCLES	3572	-
	RPU_EXT_MEM_REQUESTS	60	∼64
	RPU_DCACHE_WBACK	0	0
10	L1D_CACHE	711	640+60+14
	L1I_CACHE_REFILL	10	∼10
	L1D_CACHE_REFILL	637	640
	CPU_CYCLES	34944	-
	RPU_EXT_MEM_REQUESTS	636	∼640
	RPU_DCACHE_WBACK	0	0
100	L1D_CACHE	7014	6400+600+14
	L1I_CACHE_REFILL	10	∼10
	L1D_CACHE_REFILL	5457	6400
	CPU_CYCLES	300378	-
	RPU_EXT_MEM_REQUESTS	9917	>6400
	RPU_DCACHE_WBACK	4451	∼5376
1000	L1D_CACHE	70044	64000+6000+14
	L1I_CACHE_REFILL	10	∼10
	L1D_CACHE_REFILL	48489	64000
	CPU_CYCLES	2715226	-
	RPU_EXT_MEM_REQUESTS	96893	>64000
	RPU_DCACHE_WBACK	47832	∼62976

Table 6.10: L1 writes cache misses - R5_0

6.2 Experiments with contenders

The single-core tests on the Cortex A53 and Cortex R5 were performed step by step. From the results obtained with such tests, it was verified that each microbenchmark is working as expected, forcing the target cache to perform specific operations, and some unexpected behaviors of the platform were revealed. For this reason, it's possible to perform several experiments with such microbenchmarks when all the cores in each cluster of the target platform are turned on, thus potentially contending for the shared hardware resources.
In this section, firstly the different experiments performed will be explained in detail. Afterwards, results collected from the PMCs will be summarized and discussed comparing them with the ones obtained with the tests performed on single core mode.

6.2.1 List of experiments

Several experiments were defined to be executed for the Zynq UltraScale EG+ board. In particular, 36 experiments were performed, noting that:

1. Experiments from 1 to 12 are aimed to collect the results from the PMCs of the main core of the RPU cluster, which is **R5_0** of Cortex R5 processors. Those experiments are listed in table (6.11).

2. For what concerns the remaining experiments, they collect results of the PMCs related to the main core of the Cortex A53 cluster, which is **A53_0**. In this case, 24 experiments were performed, which are listed in two different tables in order to distinguish the ones focused just on load instructions (6.12) and the ones aimed to perform just store operations (6.13).

6.2.2 Task analysis: main core of the Cortex R5 cluster

Considering the experiments from number 1 to 12 (table 6.11), it's possible to note that the first 6 ones focus on load instructions, whereas the other ones focus on store instructions.
In each cell of such table, there are written the abbreviations of the names of the microbenchmarks implemented in each core for the specific experiment to be performed. Therefore, it's important to highlight what they are aimed to do:

- `LoadL1` is the microbenchmark that has to generate high workload on the L1 data cache using just load instructions. This means that it is focused on memory read operations and no cache misses will be experienced (so load hits).

- The microbenchmark named as `LoadMem` aims to cause systematic cache misses in the L1 data cache, forcing the system to perform just memory read operations at the main memory. Therefore, L1 data cache has to fetch the data always from the main memory (so load misses), meaning that such microbenchmark suffers and causes high contention.

- `StoreL1` is the algorithm that has to cause cache hits in L1 data cache using store instructions. This means that the same data will be written in specific memory addresses that can be completely mapped in the cache lines of the L1 data cache.

- The last microbenchmark employed for these experiments is called `StoreMem` and its goal is to generate cache misses in L1 data cache with store operations (store misses). In fact, this microbenchmark will write data in many memory addresses in such a way that they cannot be mapped in the first level cache and eviction will be experienced for each store instruction. Note that this microbenchmark uses a very large array size (2 MB) instead of the 64 kB one used in the last set

of results shown before so that roughly no store hits occur despite the pseudo-random replacement policy in the L1 data cache.

The descriptions given about the aforementioned microbenchmarks give the possibility to understand the goal of each experiment. In the first experiment, for instance, all the cores are running at the same time the `LoadL1` microbenchmark, meaning that the analysis aims to observe if the task to be handled by the main core of Cortex R5 cluster is affected by the actions of the other cores. In fact, since that no cache misses should be experienced, the data handled in each core has to be kept in each local L1 cache. Therefore, no contention should happen and this can be noted with the results obtained with the PMCs of the target processor architecture.
In general, experiments have been built to test the impact of contention incrementally. For instance `exp1` keeps all activities local in L1. `exp2` makes the neighbor core in the same cluster to create contention, keeping the main core with local accesses in L1. `exp3` raises the level of contention with memory accesses from the cores in the other cluster. Then, `exp4`, `exp5` and `exp6` are analogous, but with the main core accessing memory sustainedly so that it should suffer the impact of contention.

6.2.3 Task analysis: main core of the Cortex A53 cluster

For what concerns the remaining 24 experiments, they are aimed to observe the dynamics of the main core in the APU of the target Zynq board when load instructions (table 6.12) and store instructions (table 6.13) are performed.
Each cell of the aforementioned tables represents the microbenchmarks employed in each core for each experiment, as explained in section (6.2.2) for table (6.11). Note that the same (conceptual) microbenchmarks are used also for these cases.
Two additional microbenchmarks devised for the L2 cache of Cortex A53 processors are included in the task analysis of the Cortex A53_0 core, which

Task analysis - Main core R5_0						
#	**R5_0**	R5_1	A53_0	A53_1	A53_2	A53_3
Exp1	**LoadL1**	LoadL1	LoadL1	LoadL1	LoadL1	LoadL1
Exp2	**LoadL1**	LoadMem	LoadL1	LoadL1	LoadL1	LoadL1
Exp3	**LoadL1**	LoadMem	LoadMem	LoadMem	LoadMem	LoadMem
Exp4	**LoadMem**	LoadL1	LoadL1	LoadL1	LoadL1	LoadL1
Exp5	**LoadMem**	LoadMem	LoadL1	LoadL1	LoadL1	LoadL1
Exp6	**LoadMem**	LoadMem	LoadMem	LoadMem	LoadMem	LoadMem
Exp7	**StoreL1**	StoreL1	StoreL1	StoreL1	StoreL1	StoreL1
Exp8	**StoreL1**	StoreMem	StoreL1	StoreL1	StoreL1	StoreL1
Exp9	**StoreL1**	StoreMem	StoreMem	StoreMem	StoreMem	StoreMem
Exp10	**StoreMem**	StoreL1	StoreL1	StoreL1	StoreL1	StoreL1
Exp11	**StoreMem**	StoreMem	StoreL1	StoreL1	StoreL1	StoreL1
Exp12	**StoreMem**	StoreMem	StoreMem	StoreMem	StoreMem	StoreMem

Table 6.11: Task analysis for main core of Cortex R5 cluster

are exclusively used for Cortex A53 processors since L2 cache is present just in the APU:

- `LoadL2` is the one that aims to cause cache hits in the L2 cache through the use of load instructions. Therefore, cache misses are experienced in L1 data cache and no information will be fetched from the main memory.
- The last microbenchmark introduced in the experiments from number 25 to 36 is called `StoreL2`. The goal of this algorithm is to force the storing of specific data in L2 cache, causing cache misses in L1 and cache hits in L2. Therefore, after some iterations, such data will not be fetched from the main memory as it happens in the case of the `LoadL2` microbenchmark and it will be found always in L2 data cache.

Due to the presence of the L2 data cache that is a hardware resource shared among the four cores of the Cortex A53 cluster, it's needed to perform more experiments in order to consider the different scenarios that can occur in

avionics and/or automotive systems when consolidating some critical tasks. Therefore, L2 data cache has to be stressed like the L1 data one.
The goal of each experiment is quite similar to that of the experiments from number 1 to 12 performed for R5_0 (see Table 6.11). The experiment number 16, for instance, is almost equal to experiment number 3, since that each core is executing the `LoadMem` microbenchmark, except the main core, which executes the `LoadL1` microbenchmark. These experiments are different just because the target core is not R5_0 but A53_0, meaning that `LoadL1` is executed now by this last one.
The experiments from number 17 to 20 and from 29 to 32, for instance, are different from the previous ones. In fact, both sets of experiments aim at forcing cache hits in the L2 data cache for the main core of the APU when there are contenders that can create contention. Note that such contenders are all the other processors, namely the three cores of Cortex A53 and the two cores of Cortex R5 clusters.

Task analysis - Main core A53_0						
#	R5_0	R5_1	**A53_0**	A53_1	A53_2	A53_3
Exp13	LoadL1	LoadL1	**LoadL1**	LoadL1	LoadL1	LoadL1
Exp14	LoadL1	LoadL1	**LoadL1**	LoadL2	LoadL2	LoadL2
Exp15	LoadL1	LoadL1	**LoadL1**	LoadMem	LoadMem	LoadMem
Exp16	LoadMem	LoadMem	**LoadL1**	LoadMem	LoadMem	LoadMem
Exp17	LoadL1	LoadL1	**LoadL2**	LoadL1	LoadL1	LoadL1
Exp18	LoadL1	LoadL1	**LoadL2**	LoadL2	LoadL2	LoadL2
Exp19	LoadL1	LoadL1	**LoadL2**	LoadMem	LoadMem	LoadMem
Exp20	LoadMem	LoadMem	**LoadL2**	LoadMem	LoadMem	LoadMem
Exp21	LoadL1	LoadL1	**LoadMem**	LoadL1	LoadL1	LoadL1
Exp22	LoadL1	LoadL1	**LoadMem**	LoadL2	LoadL2	LoadL2
Exp23	LoadL1	LoadL1	**LoadMem**	LoadMem	LoadMem	LoadMem
Exp24	LoadMem	LoadMem	**LoadMem**	LoadMem	LoadMem	LoadMem

Table 6.12: Task analysis for main core of Cortex A53 cluster - Load instructions

Task analysis - Main core A53_0						
#	R5_0	R5_1	**A53_0**	A53_1	A53_2	A53_3
Exp25	StoreL1	StoreL1	**StoreL1**	StoreL1	StoreL1	StoreL1
Exp26	StoreL1	StoreL1	**StoreL1**	StoreL2	StoreL2	StoreL2
Exp27	StoreL1	StoreL1	**StoreL1**	StoreMem	StoreMem	StoreMem
Exp28	StoreMem	StoreMem	**StoreL1**	StoreMem	StoreMem	StoreMem
Exp29	StoreL1	StoreL1	**StoreL2**	StoreL1	StoreL1	StoreL1
Exp30	StoreL1	StoreL1	**StoreL2**	StoreL2	StoreL2	StoreL2
Exp31	StoreL1	StoreL1	**StoreL2**	StoreMem	StoreMem	StoreMem
Exp32	StoreMem	StoreMem	**StoreL2**	StoreMem	StoreMem	StoreMem
Exp33	StoreL1	StoreL1	**StoreMem**	StoreL1	StoreL1	StoreL1
Exp34	StoreL1	StoreL1	**StoreMem**	StoreL2	StoreL2	StoreL2
Exp35	StoreL1	StoreL1	**StoreMem**	StoreMem	StoreMem	StoreMem
Exp36	StoreMem	StoreMem	**StoreMem**	StoreMem	StoreMem	StoreMem

Table 6.13: Task analysis for main core of Cortex A53 cluster - Store instructions

6.2.4 Final results and Research Observations

Thanks to the use of Bash scripts, which were specifically devised for this set of experiments, many results were collected with all relevant PMCs of main core, regardless of the cluster where the task under analysis was run, and some of them were used to compute the number of cycles per instruction that it took to run each experiment. Since the number of experiments is very large and hence, the number of counters is also huge, only the results of the CPU_CYCLES and INST_RETIRED counters are shown in table (6.14). Therefore, it was possible to obtain the final results shown in table (6.15), which are represented also in figure (6.1) for the sake of convenience. Note that each experiment was performed disabling firstly the data prefetcher of all the cores, including the target main core under analysis, to avoid uncontrolled behavior of any core.

In this last table, it's important to highlight that `CPI` stands for *Cycles Per Instruction* and that `Mem` stands for the main memory. Moreover, all the experiments are categorized in such a way that the reader can understand

better the type of microbenchmark studied in the main core, which memory level they are stressing, the type of instructions used (Load or Store) and the target processing cluster.
The `CPI` values were obtained dividing the counter of the CPU cycles of the target main core (CPU_CYCLES) by the number of instructions executed in the corresponding experiment (INST_RETIRED).

Experiment	CPU_CYCLES	INST_RETIRED
Exp 1	394118	130240
Exp 2	394528	130240
Exp 3	394647	130240
Exp 4	8889679	130240
Exp 5	9049911	130240
Exp 6	9174290	130240
Exp 7	534830	133580
Exp 8	539924	133580
Exp 9	567259	133580
Exp 10	4579026	133313
Exp 11	5082107	133313
Exp 12	12047451	133313
Exp 13	1287787	1282160
Exp 14	1297474	1282160
Exp 15	1305180	1282160
Exp 16	1290713	1282160
Exp 17	4639647	130103
Exp 18	5139346	130103
Exp 19	4832349	130103
Exp 20	4818012	130103
Exp 21	16545149	130102
Exp 22	18928491	130102
Exp 23	17066002	130102
Exp 24	19619430	130102
Exp 25	189313	133609
Exp 26	190034	133609
Exp 27	199632	133609
Exp 28	199599	133609
Exp 29	859128	133251
Exp 30	2045853	133251
Exp 31	1131668	133251
Exp 32	1197269	133251
Exp 33	4268173	133112
Exp 34	5624223	133112
Exp 35	8723100	133112
Exp 36	8361755	133112

Table 6.14: CPU cycles and instructions performed in each experiment

Processing unit	Instructions	Microbenchmark	Experiment	CPI
RPU	Load	L1	Exp 1	3.03
			Exp 2	3.04
			Exp 3	3.04
		Mem	Exp 4	68.26
			Exp 5	69.49
			Exp 6	70.45
	Store	L1	Exp 7	4.01
			Exp 8	4.06
			Exp 9	4.27
		Mem	Exp 10	34.35
			Exp 11	38.13
			Exp 12	90.38
APU	Load	L1	Exp 13	1.00
			Exp 14	1.01
			Exp 15	1.02
			Exp 16	1.01
		L2	Exp 17	35.67
			Exp 18	39.51
			Exp 19	37.15
			Exp 20	37.04
		Mem	Exp 21	127.18
			Exp 22	145.50
			Exp 23	131.18
			Exp 24	150.81
	Store	L1	Exp 25	1.42
			Exp 26	1.43
			Exp 27	1.51
			Exp 28	1.51
		L2	Exp 29	6.45
			Exp 30	15.36
			Exp 31	8.51
			Exp 32	9.00
		Mem	Exp 33	32.07
			Exp 34	42.26
			Exp 35	65.55
			Exp 36	62.83

Table 6.15: Cycles per instruction results

Focusing on the plot represented in figure (6.1), instead, it's possible to comment the results in a easier better way. As expected, the CPI values are higher in the experiments that are forcing the specific processing unit to access the main memory, like the ones from number 4 to 6 or from number 21 to 24. In fact, even though the number of instructions performed in the experiments from number 1 to 3 is the same also in the experiments from 4 to 6, the number of execution cycles is quite bigger.

Cortex-R0 results

First of all, we observe that the CPI for experiments accessing only L1 cache are virtually insensitive to contention. For instance, in the case of loads hits, experiments 1-3 have roughly the same CPI, and so it is the case for experiments 7-9 for store hits. Note that, while this behavior is expected, it is not always the case in all architectures since cache snooping and cache inclusion characteristics may lead to some interference even if the task under analysis does not use any shared resource.

Observation 1: *Load hits and store hits are insensitive to contention. Hence, critical tasks operating mostly with local data can be consolidated with any other software without specific constraints.*

In the case of load misses, we observe that the CPI is also nearly constant regardless of contention. A closer look at the results reveals, as pointed out before, that load frequency is lower than store frequency. Thus, load misses stall execution in the core which, in the case of contenders, prevents them from creating higher contention. Hence, the task under analysis, although it experiences some relevant contention, does not suffer a noticeable increase in its CPI since the intrinsic parallelism of the memory system allows serving multiple load requests from the different cores in parallel. In fact, in the case of experiment 6 a very slight CPI increase is observed when memory must

serve load requests from all cores simultaneously. We, therefore, identify a key information for task consolidation:

Observation 2: *Load misses cause limited contention in practice, so critical tasks can be consolidated with programs with such profile without experiencing a relevant increase in their execution time.*

When analyzing experiments 10-12, thus for store misses, we realize that, as shown before, CPI without contention is lower than for loads. However, those tasks are much more sensitive to contention that load-based ones. For instance, when adding a store miss contender (exp 11), CPI of the main core increases by more than 10%. When placing store miss contenders in all other cores, the CPI grows by a factor of 2.6x (so 160% higher CPI). The reason behind this behavior is that store misses perform frequent accesses to memory due to being processed at a higher rate and due to causing additional accesses for dirty cache line evictions. Thus, this leads to the third key observation:

Observation 3: *Store misses cause high contention in memory, so critical tasks sensitive to contention must not be consolidated with store-miss programs as contenders.*

Cortex-A53 results

As for the R0 cores, load-hit and store-hit tasks are highly insensitive to contention. This can be seen comparing experiments 13-16 (load hits) as well as 25-28 (store hits). As shown, the CPI remains almost constant regardless of the contention caused by the tasks in the other cores, whose impact is negligible regardless of whether they access their local L1 caches, the shared L2 cache or main memory. Hence, similar conclusions can be reached for load-hit and store-hit programs in the A53 and R0 cores:

Observation 4: *Load hits and store hits are insensitive to contention, regardless whether it occurs in the L2 cache or in memory. Hence, critical tasks operating mostly with local data can be consolidated with any other software without specific constraints also in the A53 cores.*

Regarding experiments targeting L2 cache, they have been devised so that their data fits comfortably in L2 and hence, contention can only occur in the access ports and queues, but not due to mutual evictions. In practice, mutual evictions can be avoided using cache partitioning, which was not considered in the scope of this Master thesis for the sake of simplicity.
Experiments 17-20 evaluate the case for load L2 hits. We observe that making contenders access also the L2 cache frequently (exp 18 compared w.r.t. exp 17) leads to a CPI increase of around 10%. Instead, if contenders access main memory (exp 19-20), their L2 access frequency is lower and hence, their contention is lower (below 5%). Hence, load L2 hit tasks are not very sensitive to contention and, only contention in the access to L2 is relevant. Hence, we raise the following observation for software consolidation:

Observation 5: *Load L2 hits are highly insensitive to contention and only they may suffer some little contention if contenders access L2 frequently. Hence, Load L2 hit tasks may be better consolidated with tasks keeping data locally in L1 or accessing mostly memory, but integrating them with L2-hungry tasks has limited effects.*

When analyzing the case of store L2 hits (experiments 29-32), we observe similar trends but with much larger magnitude. In particular, store L2 hits can make CPI grow above 2x due to contention, whereas memory traffic, which creates lower L2 contention, can still increase the CPI of the task under analysis by 30-40%. Overall, this exposes two key facts: store L2 hits create very high contention and are very sensitive to contention. The

main reason for this behavior is the fact that stores are normally processed asynchronously, so that the latency of store L2 hits can be normally hidden with some queues and buffers in L1 and L2 caches. However, as long as those buffers and queues get saturated due to contention, back pressure is created and execution in the core gets stalled, thus leading to significant relative CPI increases. Hence, we raise the following observations:

Observation 6: *Store L2 hit critical tasks must not be consolidated with tasks creating high L2 contention. Hence, they can be consolidated with tasks keeping their data locally in L1 or, at most, with load misses tasks that, despite creating some contention in the L2, such contention is limited.*

Observation 7: *Store L2 hit tasks have the ability to increase contention in L2 dramatically. Hence, they are compatible with critical tasks as long as those tasks are insensitive to L2 contention, either because they keep their data locally in L1 or because they already experience high latency accessing main memory so that the relative impact of L2 contention is low.*

Load misses (experiments 21-24) have already a high CPI, so the relative impact of contention is low. Still, such impact is relevant when contenders access L2 cache or memory frequently, which may increase the CPI by around 14-19%. Thus, the following observation holds:

Observation 8: *Load misses are sensitive to contention to some extent. Hence, consolidating load misses critical tasks with tasks that mostly access L1 globally is the most convenient solution. Still, having some L2 or memory intensive tasks running together has limited effects.*

Finally, store misses (experiments 33-36) are highly sensitive to contention, especially if such contention occurs in memory. In particular, store L2 hits cause a 30% increase in CPI, whereas store misses contenders may make the

CPI of the task under analysis grow by 2x. Thus, we observe the following:

Observation 9: *Store misses are highly sensitive to contention, especially if such contention occurs in memory. Hence, store misses critical tasks must be consolidated with tasks causing low contention such as, for instance, L1 hit tasks and those accessing shared resources not too often (e.g. load misses).*

Observation 10: *Store misses create very high contention in memory. Therefore, critical tasks running together with this type of contenders must keep their data locally in L1 as much as possible to keep their CPI low.*

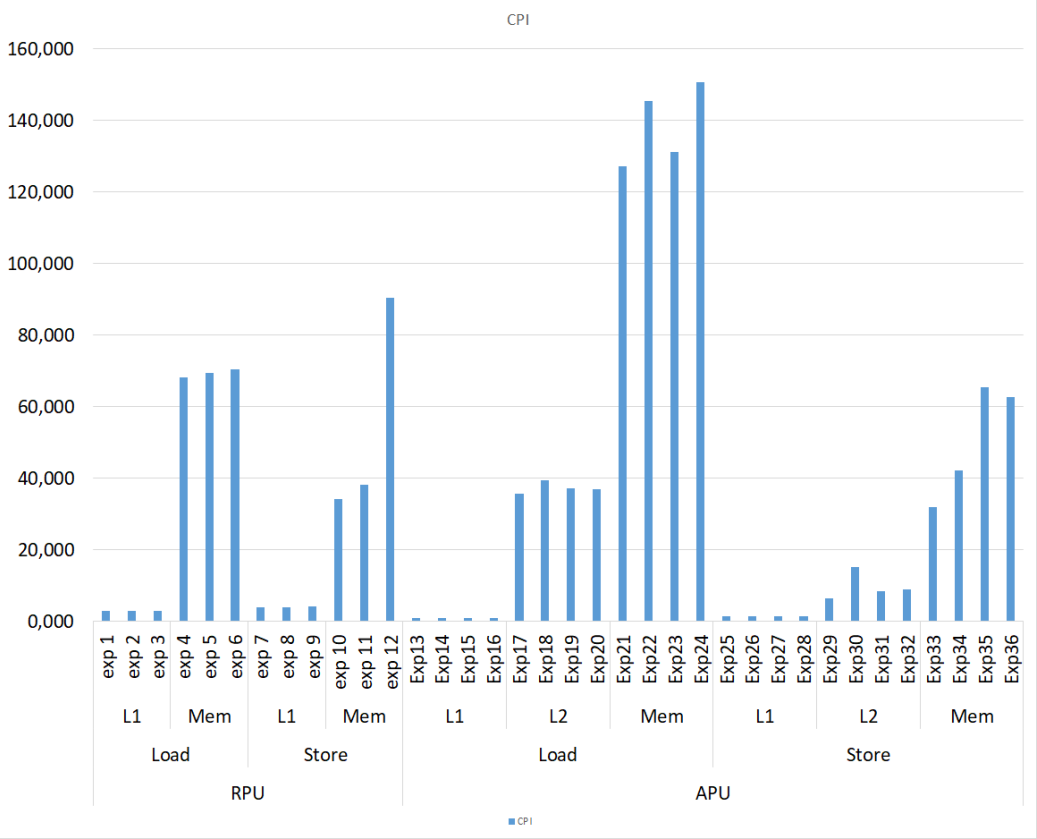

Figure 6.1: Cycles per instruction plot of the experiments with contenders

Chapter 7

Budget

In this chapter, it will be presented the different elements to be considered to estimate the budget for this Master thesis. Firstly, the costs for the devices employed for the experiments and for the people involved in the project are presented, then the financial viability of the approach proposed is discussed.

7.1 Costs

- It was needed to buy the Zynq UltraScale+ EG board in order to implement and evaluate the microbenchmarks devised for this Master thesis. The cost of purchasing such board is around 2,500€.

- The cost per hour to be incurred for the development of this work by myself is estimated in 9€. Therefore, given that the Master thesis work required around 900, which is in line with the 30 × 25-to-30 total hours planned for a Master project of 30 ECTS, with 25-30 hours per ECTS, my personnel cost for this master is 8,100€. Note that such cost includes parts that, for the development of the technology only would not be incurred (i.e. reading bibliography, preparing this thesis, preparing the presentation). Still, for the sake of simplicity, these costs are not broken down.

- Members of the CAOS team at BSC spent also some of their time for this work, in particular my advisor Jaume Abella and the engineer Mike Fernández. Their effort is estimated to be around 10% of the time devoted by myself. Also, on average, the cost per hour assumed for people involved in this work is estimated to be twice my cost, so 18€ per hour. Hence, this leads to 1,620€.

Overall, the total cost for the development of this technology amounts 12,220€.

Budget	End of February - Middle of July	Personal cost	∼8100€
		BSC cost	∼1620€
		Zynq platform	∼2500€

Table 7.1: Pay and cost per hour for Master thesis

7.2 Financial viability

For what concerns the financial viability, note that any company willing to use the technology developed in this Master thesis will already need the Zynq UltraScale+ EG platform for the system to be deployed. Hence, such cost would not be incurred by the developed technology itself. Instead, the use of this microbenchmark technology requires instrumenting software tasks of the end user to collect information on the Zynq board and reach conclusions on how to consolidate software in the system to be deployed. For that purpose, we assume that the cost to perform such work for just one task is at most 1 hour. Hence, assuming the cost of a skilled engineer, such cost could be up to 18€. This leads to the conclusion that the use of this technology has a very low cost w.r.t. the amount of money spent for the board and the development and validation of the software to be deployed. In fact, with this estimation, and assuming that the system to be deployed will consist typically in a number of tasks in the order of some tens or up to few hundreds of tasks,

the cost of using this technology is really low (e.g. 3,600€ for a 200-task system).

Chapter 8

Conclusions

The adoption of high-performance hardware platforms such as the Zynq UltraScale+ EG one on critical real-time embedded systems is a need in domains such as automotive and avionics among others. This work has shown, through the integration and adaptation of a measurement-based methodology based on microbenchmarks, that such platform brings some challenges due to the interference that cores can experience when accessing shared hardware resources. Therefore, software consolidation must be performed carefully so that hardware is exploited efficiently, particularly for critical real-time tasks. The work in this thesis analyzes in detail how contention occurs when accessing different shared hardware resources, such as shared caches and main memory, across different computing clusters, namely the real-time Cortex-R5 one and the high-performance Cortex-A53 one, and for different operation types (loads and stores).

Our results bring highly valuable conclusions in the form of observations, which are key to allow end users perform appropriate task consolidation in the system for its deployment. In particular, our observations indicate what tasks should execute concurrently and what task types must not do it, so that contention experienced is limited, thus leading to an efficient use of shared hardware resources by experiencing limited contention. Ultimately, this provides evidence that WCET estimates for tasks in isolation only grow

slightly due to contention, so that they can still finish by their respective deadlines. Hence, the observations in this thesis are a key source of information for end users willing to use this platform for critical real-time embedded systems, since they can build on those observations to properly schedule tasks.

8.1 Future development

The research findings presented in this thesis open the door to a number of research paths:

- Task scheduling. While observations are provided in the form of guidelines for scheduling purposes, scheduling algorithms building upon profiled information from tasks (with PMCs) can be build on top of our observations so that efficient schedules are devised automatically, thus minimizing (or even removing) user intervention.

- Contention models. While our analysis reveals the magnitude of execution time increase incurred due to contention, actual (fully-reliable) bounds have not beed completely devised. By further testing the platform, those reliable bounds can be devised and analytical contention models can be devised so that impact of contention on a given task can be reliably and tightly estimated without having it to run concurrently with other tasks of the system. This can be a very valuable input for task scheduling so that scheduling decisions are based on actual predictions rather than on qualitative observations.

- Other components. The target of the analysis in this thesis has been general purpose processor cores. However, the platform includes other computing resources such as a GPU and an FPGA. Analyzing the contention experienced by those components remains as future work to enable their effective use while having guarantees about the impact of contention on execution time also for those components.

Bibliography

[1] Gabriel Alejandro Fernandez Diaz, *"Enhancing Timing Analysis for COTS Multicores for Safety-Related Industry: a Software Approach"*. PhD thesis, Universitat Politecnica de Catalunya, 2018.

[2] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The Worst-case Execution-time Problem—Overview of Methods and Survey of Tools," *ACM Trans. Embed. Comput. Syst.*, vol. 7, pp. 36:1–36:53, May 2008.

[3] J. Abella, C. Hernandez, E. Quiñones, F. J. Cazorla, P. R. Conmy, M. Azkarate-askasua, J. Perez, E. Mezzetti, and T. Vardanega, "WCET analysis methods: Pitfalls and challenges on their trustworthiness," in *10th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pp. 1–10, June 2015.

[4] E. Mezzetti and T. Vardanega, "On the industrial fitness of WCET analysis," in *Worst-Case Execution Time (WCET) Analysis Workshop*, 2011.

[5] Rapita Systems Ltd., "RapiTime - worst case execution time (WCET) analysis for critical systems. https://www.rapitasystems.com/products/rapitime," 2018.

[6] M. Fernández, R. Gioiosa, E. Quiñones, L. Fossati, M. Zulianello, and F. J. Cazorla, "Assessing the Suitability of the NGMP Multi-core Pro-

cessor in the Space domain," *EMSOFT' 12*, pp. 175–184, October 7, 2012.

[7] ARM, *ARM Cortex-A Series: Programmer's Guide for ARMv8-A*, 2015.

[8] L. Kosmidis, J. Abella, E. Quiñones, and F. J. Cazorla, "A Cache Design for Probabilistically Analysable Real-time Systems," in *Design Automation Test in Europe (DATE) Conference*, 2013.

[9] SPARC, *The SPARC Architecture Manual*, 1991,1992.

[10] András Vajda, *Programming Many Core Chips*, 2011.

[11] P. Radojković, S. Girbal, A. Grasset, E. Quiñones, S. Yehia, and F. J. Cazorla, "On the Evaluation of the Impact of Shared Resources in Multithreaded COTS Processors in Time-critical Environments," *ACM Trans. Archit. Code Optim.*, vol. 8, pp. 34:1–34:25, Jan. 2012.

[12] G. Fernandez, J. Jalle, J. Abella, E. Quiñones, T. Vardanega, and F. J. Cazorla, "Resource Usage Templates and Signatures for COTS Multicore Processors," in *Proceedings of the 52Nd Annual Design Automation Conference*, DAC '15, (New York, NY, USA), pp. 155:1–155:6, ACM, 2015.

[13] E. Díaz, E. Mezzetti, L. Kosmidis, J. Abella, and F. J. Cazorla, "Modelling Multicore Contention on the AURIXTM TC27x," in *Proceedings of the 55th Annual Design Automation Conference*, DAC '18, (New York, NY, USA), pp. 97:1–97:6, ACM, 2018.

[14] "SAFURE - Safety And Security By Design For Interconnected Mixed-Critical Cyber-Physical Systems. https://safure.eu," 2018.

[15] G. Fernandez, F. J. Cazorla, and J. Abella, "Consumer Electronics Processors for Critical Real-Time Systems: a (Failed) Practical Experience," in *9th European Congress on Embedded Real Time Software and Systems (ERTS^2)*, 2018.

[16] J. Nowotsch and M. Paulitsch, "Leveraging Multi-core Computing Architectures in Avionics," in *Proceedings of the 2012 Ninth European Dependable Computing Conference*, EDCC '12, (Washington, DC, USA), pp. 132–143, IEEE Computer Society, 2012.

[17] J. Nowotsch, M. Paulitsch, D. Bühler, H. Theiling, S. Wegener, and M. Schmidt, "Multi-core Interference-Sensitive WCET Analysis Leveraging Runtime Resource Capacity Enforcement," in *2014 26th Euromicro Conference on Real-Time Systems (ECRTS)*, pp. 109–118, July 2014.

[18] I. Agirre, J. Abella, M. Azkarate-Askasua, and F. J. Cazorla, "On the tailoring of CAST-32A certification guidance to real COTS multicore architectures," in *2017 12th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pp. 1–8, June 2017.

[19] S. Girbal, J. Le Rhun, and H. Saoud, "METrICS: a Measurement Environment for Multi-Core Time Critical Systems," in *9th European Congress on Embedded Real Time Software and Systems (ERTS^2)*, 2018.

[20] D. Griffin, B. Lesage, I. Bate, F. Soboczenski, and R. I. Davis, "Forecast-based Interference: Modelling Multicore Interference from Observable Factors," in *Proceedings of the 25th International Conference on Real-Time Networks and Systems*, RTNS '17, (New York, NY, USA), pp. 198–207, ACM, 2017.

[21] G. Fernandez, J. Abella, E. Q. nones, L. Fossati, M. Zulianello, T. Vardanega, and F. J. Cazorla, "Seeking Time-Composable Partitions of Tasks for COTS Multicore Processors," in *2015 IEEE 18th International Symposium on Real-Time Distributed Computing*, pp. 208–217, April 2015.

[22] K. Ganesan, J. Jo, W. L. Bircher, D. Kaseridis, Z. Yu, and L. K. John, "System-level Max Power (SYMPO): A Systematic Approach for Escalating System-level Power Consumption Using Synthetic Benchmarks,"

in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, (New York, NY, USA), pp. 19–28, ACM, 2010.

[23] NXP Semiconductors, *Measuring Current in i.MX Applications. Application Note, Document Number: AN5381, https://www.nxp.com/docs/en/application-note/AN5381.pdf*, 2016.

[24] ARM, *Cortex-R5: Technical Reference Manual*, 2010-2011.

[25] P. Benedicte, C. Hernandez, J. Abella, and F. J. Cazorla, "Rpr: A random replacement policy with limited pathological replacements," in *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, SAC '18, (New York, NY, USA), pp. 593–600, ACM, 2018.

[26] M. Goossens, F. Mittelbach, and A. Samarin, *The LaTeX Companion*. Reading, Massachusetts: Addison-Wesley, 1993.

[27] D. Knuth, "Knuth: Computers and typesetting."

[28] XILINX, *Zynq UltraScale+ Device: Technical Reference Manual*, December 22, 2017.

[29] ARM, *ARM Cortex-A53 MPCore Processor: Technical Reference Manual*, 2016.

[30] ARM, *ARM Cortex-R Series: Programmer's Guide*, 2014.

[31] ARM, *ARM Architecture Reference Manual: ARMv8 for ARMv8-A architecture profile*, December 19 2017.

[32] Wikipedia, "CPU Cache."